Philipps–Universität Marburg
Fachbereich Mathematik und Informatik

Basiskomponenten von XML Datenbanksystemen

Dissertation
zur Erlangung des Doktorgrades
der Naturwissenschaften
(Dr. rer. nat.)

dem Fachbereich Mathematik und Informatik
der Philipps-Universität Marburg
vorgelegt von

Martin Schneider
aus Marburg

Marburg/Lahn 2004

Vom Fachbereich Mathematik und Informatik
der Philipps-Universität Marburg
als Dissertation am 26. Oktober 2004 angenommen.

Erstgutachter: Prof. Dr. Bernhard Seeger

Zweitgutachter: Prof. Dr. Manfred Sommer

Tag der mündlichen Prüfung am 28. Oktober 2004

Zusammenfassung

Für die Entwicklung von vielen kleinen und großen Softwaresystemen reichen herkömmliche (objekt-)relationale Datenbanksysteme nicht mehr aus. Viele interessante Daten sind in der Praxis nicht voll strukturiert und somit nicht effektiv mit einem Standarddatenbanksystem zu verwalten. Es werden deshalb neuartige standardisierte Systeme für unstrukturierte bzw. semi-strukturierte Daten benötigt.

Diese Lücke wird nun von *nativen XML Datenbanksystemen* geschlossen, die als Datenformat das vom W3C standardisierte XML verwenden. XML Datenbanksysteme unterstützen außerdem viele weitere XML Standards, wie beispielsweise XSchema für Grammatiken, XPath und XQuery für die Anfrageverarbeitung, XSLT für Transformationen und DOM und SAX für die Applikationsanbindung.

In dieser Arbeit werden Grundlagen von nativen XML Datenbanksystemen betrachtet, sowie neue Strukturen vorgeschlagen und alte Strukturen optimiert. Es wird auf eine solide Basis zum Testen von Algorithmen Wert gelegt. Hierzu wurde ein Testframework innerhalb der Java-Bibliothek XXL implementiert und anschließend verwendet.

Die XXL Bibliothek enthielt bereits vor dieser Arbeit einige Komponenten, die für die Implementierung von Datenbanksystemen eingesetzt werden konnten, beispielsweise eine generische Anfrageverarbeitung und Indexstrukturen. Zusätzlich zu den vorhandenen Komponenten wurden nun neue hinzugefügt, so z. B. eine Komponente für den direkten Festplattenzugriff, ein frei konfigurierbarer Recordmanager, sowie ein Datenbank-Framework.

Das zentrale Anliegen der Arbeit ist die Optimierung der Speicherungsebene von nativen XML Datenbanksystemen. Wichtig ist, dass bei der Abbildung von XML Dokumenten auf den Externspeicher die Baumstruktur erhalten bleibt und somit eine performante Anfragenverarbeitung mit wenigen Externspeicherzugriffen möglich wird. Ähnlich wie bei R-Bäumen, können für XML Speicherungsstrukturen verschiedene Splitalgorithmen angegeben werden, die gewisse Heuristiken verfolgen. Hier zeigte sich der neu entwickelte, so genannte *One-*

CutSplit mit Scaffold als klar überlegen gegenüber den bisher bekannten Splitalgorithmen aus der Literatur.

Für das Einfügen von Dokumenten wurde weiterhin ein Bulkloading Mechanismus implementiert. Es konnte gezeigt werden, dass die Speicherstruktur für die hiermit erzeugten Dokumente deutlich besser war als bei der Benutzung von Splitalgorithmen. Dies macht sich erheblich in den Antwortzeiten von Anfragen bemerkbar.

Zur Beschleunigung der Anfrageverarbeitung sind in nativen XML Datenbanksystemen Indexstrukturen unverzichtbar. Zu diesem Zweck wurde ein neuartiger Signaturindex entwickelt und in die XML Speicherungsstruktur unter Verwendung von Aggregaten integriert. Die Evaluierung des Indexes zeigte einen deutlichen Vorteil bei der Auswertung von XPath-Ausdrücken.

Weiterhin konnten erstmals durch die Benutzung des Datenbank-Frameworks von XXL native Speicherungsverfahren für XML mit solchen verglichen werden, die auf relationalen Datenbanksystemen aufsetzen. Hierbei zeigte sich, dass nativer XML Speicher auch bei einfachen XPath-Anfragen gute Leistungswerte besitzt. Bei Navigations- und Änderungsoperationen ist der native XML Speicher den relationalen Verfahren deutlich überlegen.

In der Anfrageverarbeitung auf XML Daten spielen allerdings nicht nur XPath und XQuery eine Rolle. Für die Bearbeitung von großen Mengen von XML Dokumenten sind Operatoren sinnvoll, welche eine Verarbeitung durch Abbildung von XML Dokumenten auf neue XML Dokumente realisieren. Dies ist analog zur relationalen Algebra, in der allerdings der Grunddatentyp Tupel Verwendung findet. Im Vergleich zum relationalen Modell werden für XML jedoch viele verschiedene Operatoren benötigt, die nicht auf wenige Grundoperationen zurückgeführt werden können.

In dieser Arbeit werden einige neue Operatoren vorgestellt, die nicht nur für die Anfrageverarbeitung innerhalb von XML Datenbanksystemen, sondern auch für Anfragen im Internet geeignet sind. Durch das entwickelte Framework soll es Anwendern in Zukunft auf einfache Art und Weise möglich sein, Internetquellen in eigene Anfragen einzubauen.

Inhaltsverzeichnis

Kapitel 1

Motivation

Seitdem es Computer gibt, dienen sie dazu, mehr oder weniger große Datenmengen zu speichern und Operationen auf ihrer Basis durchzuführen. Die ersten Computeranwendungen in staatlichen Organisationen wurden in den 1950er Jahren beispielsweise für die Berechnung der individuellen Steuerlast eingesetzt. Seit den Pioniertagen der Computerindustrie sind mittlerweile gut 50 Jahre vergangen. Computer sind heute allgegenwärtig anzutreffen, ob als Server in Rechenzentren, als Arbeitsplatzrechner, als Spielkonsole im Kinderzimmer, im Handy oder im PDA. In der heutigen Informationsgesellschaft entstehen Daten und Informationen im Überfluss, und viele Daten sind von überall aus über das Internet zugreifbar. Die gegenwärtige *Informationslawine* [Gra04] trifft sowohl Privatpersonen, als auch Firmen und jede Form von Organisationen.

Die Datenverwaltung nimmt dementsprechend in den letzten Jahren einen immer höheren Stellenwert ein. Gleichzeitig wird das Publikum, das Datenverwaltungssysteme benötigt, immer größer. Hieraus ergibt sich die Notwendigkeit, dass die eingesetzte Software für Laien einfach installierbar und wartbar sein muss. Und selbst ein unerfahrener Anwender muss zufriedenstellende Ergebnisse erhalten, wenn er seine Anfragen an die Datenbasis stellt.

Traditionell ist jede Firma auf eine effiziente Datenverwaltung angewiesen. In der Vergangenheit ging es um Eingabe und Verwaltung von Aufträgen, Rechnungen und Kundendaten. Doch heute geht die Informationsverwaltung viel weiter. Für ein Unternehmen ist das Wissen der eigenen Mitarbeiter Geld wert. Das Wissen aller Mitarbeiter sollte immer abrufbereit zur Verfügung stehen, so

dass sich Planungen, wie beispielsweise neue Werbekampagnen, am Gesamtwissen der Firma orientieren können. Firmenkunden können hierdurch kompetent und *individuell* betreut werden, was wiederum zu einem Plus an neuen Aufträgen führen kann. Durch ein gutes, über traditionelles *Customer Relationship Management (CRM)* hinausgehendes Wissensmanagement können sogar neue Ideen für künftige Produkte entstehen. Dies bringt enorme Wettbewerbsvorteile mit sich, die letztlich den Fortbestand einer Firma sichern können.

1.1 Datenbanksysteme

Im Jahr 1970 wurde das relationale Modell von Codd [Cod70] vorgestellt, das bald danach relationale Datenbanksysteme (RDBS) nach sich zog. In den 1980er Jahren wurden diese durch die Einführung der standardisierten Sprache SQL [ANS86] marktbeherrschend. Ganz wesentlich für diese Systeme war die Entwicklung der Daten- und Indexstruktur des B^+-Baums [Com79], der ein effizientes Einfügen und Abfragen von Daten erlaubt. Darüber hinaus ist die Speicherung im B^+-Baum platzsparend möglich.

Seitdem hat sich wenig verändert. Mit dem Siegeszug der objektorientierten Denkweise wurden zwar viele Erweiterungen in SQL eingeführt, jedoch ist das Grundprinzip dasselbe geblieben. Echte objektorientierte Datenbanksysteme haben sich in der Praxis am Markt nicht durchsetzen können.

Was aber ist der eigentliche Sinn und Zweck von Datenbanken? Eine Datenbank für sich genommen ist i. d. R. sinnlos, da der Zugriff über SQL nur wenigen Experten vorbehalten bleibt. Für den normalen Anwender sind immer spezielle Datenbankapplikationen nötig, die ihm gewisse Funktionen über möglichst allgemein verständliche Bildschirmmasken zur Verfügung stellen. Über solche Masken kann dann beispielsweise eine Sekretärin ihre Daten eingeben und der Manager eines Unternehmens den Überblick über die Verkaufszahlen behalten.

Die Hauptnutzer und Profiteure von Datenbanksystemen sind also Applikationsentwickler, weil in der Regel nur sie direkt mit dem System arbeiten. Für sie ist ein Datenbanksystem ein wesentlicher *Building Block*, der die Speicherung, die Beantwortung von Anfragen aber auch weitere Dienste wie Mehrbe-

nutzerfähigkeit, Fehlertoleranz und Sicherheit zur Verfügung stellt. Diese wichtigen Dienste werden vom millionenfach getesteten Code eines Datenbanksystems erledigt, und zwar sicher und performant.

Doch bei all den positiven Seiten von relationalen Datenbanksystemen gibt es auch Probleme. In der Vergangenheit kam die Festschreibung des SQL-Standards leider nicht mit den schnell wachsenden Bedürfnissen der Applikationsentwickler mit. Somit mussten die Datenbankhersteller wie Oracle, IBM und Microsoft auf Druck der Kunden eigene, proprietäre Erweiterungen an SQL vornehmen. Diese Erweiterungen sind nicht standardisiert, so dass der Austausch eines Datenbanksystems gegen ein anderes inzwischen mit hohem und teilweise nicht abschätzbarem Aufwand verbunden ist.

Für Applikationsentwickler als Anwender von Datenbanksystemen gibt es jedoch noch weitere Probleme. Die Technik der relationalen Datenbanksysteme verlangt zuerst immer nach einem Schema. Dies ist bei Systemen mit vielen Anwendern bereits ein großes Problem, denn jeder Anwender hat seine eigenen Interessen. Durch viele Anwender kommen viele Interessen in die Modellwelt des Systems, und das entstehende Schema wird schnell sehr komplex. Ein vollständiges Schema, d. h. die Abbildung aller Objekte der realen Welt auf ein Datenbankschema, ist in der Praxis jedoch nicht machbar. Zusätzlich macht jeder Entitätstyp in einer Datenbank die Entwicklung einer Datenbankapplikation komplexer. In der Praxis muss hierdurch nach einem Schema gesucht werden, das mit möglichst niedrigem Entwicklungsaufwand alle Anwendergruppen zufrieden stellt.

Wenn nach dem Ende der Realisierung einer Anwendung neue Anforderungen auftreten, was der Regelfall ist, so müssen eventuell große Teile des Schemas neu überdacht werden. Eine Änderung des Schemas erfordert zusätzlich zu der Überarbeitung der Applikationen auch die Konvertierung der bereits vorhandenen Daten, was wiederum zusätzlichen Aufwand bedeutet.

1.2 (Un-)strukturierte Daten

Vor der Entscheidung eines Anwendungsprogrammierers für oder gegen die Verwendung eines Datenbanksystems muss er einige Entscheidungen treffen. Zum einen stellt sich die Frage, ob bzw. welche Dienste des Datenbanksystems benötigt werden. Zum anderen geht es ganz grundlegend darum, welche Art von Daten überhaupt verwaltet werden sollen. Nicht jede Art von Daten kann in einem traditionellen Datenbanksystem effizient verwaltet werden. In dieser Arbeit wird zwischen drei verschiedenen Arten von Daten unterschieden:

- Unstrukturierte Daten
- Semi-strukturierte Daten
- Strukturierte Daten

Eine exakte Abgrenzung der drei Termini ist schwierig. Es gibt in der Literatur verschiedene Definitionen, die sich in Details unterscheiden. Ob es sich bei konkret, in einem Szenario vorhandenen Daten um die eine oder andere Sorte Daten handelt, kann somit häufig nicht eindeutig beantwortet werden.

Bei strukturierten Daten ist vor dem Erstellen des ersten Datums dessen Schema bekannt. Zum Schema gehört, dass die interne Struktur des Datums beschrieben wird, wozu auch die Angabe von Datentypen aller Einzelteile zählt. Als Beispiel seien hier die Personaldaten der Mitarbeiter einer Firma genannt, die meistens in einer Relation eines relationalen Datenbanksystemen verwaltet werden. Der Datensatz eines Mitarbeiters gliedert sich in eine Menge von Attributen, deren Typen bekannt sind.

Allerdings können in Datenbanksystemen nicht nur voll strukturierte Daten wie die Mitarbeiterrelation gespeichert werden. Generell ist es möglich, jede Art von Information in Relationen abzulegen. Hierzu dienen die „Datentypen“ BLOB (binary large object) und CLOB (character large object). Von Typsicherheit kann bei Benutzung dieser Ablageform jedoch nicht gesprochen werden, da das Datenbanksystem selbst keinerlei Möglichkeiten hat, die Integrität

solcher Daten zu testen. Weiterhin ist die Anfragefunktionalität auf BLOB- und CLOB-Attributen nur sehr rudimentär gegeben.

Unstrukturierte Daten zu definieren, ist nicht so leicht, wie es im ersten Moment scheint. Man kann Bilder oder Klänge zu unstrukturierten Daten zählen. In beiden Fällen existieren jedoch Dateiformate, also Schemata, die diese Daten beschreiben. Weiterhin können Bilder auch textuell beschrieben werden und eine Menge von Bildern können gemeinsame Merkmale (Strukturen) besitzen. Somit enthalten Bilder implizit gewisse Informationen, die eventuell sogar strukturiert sein können und die ein Mensch ohne Mühe erfassen kann. Ein Computer hat dagegen große Mühe, mit implizit gegebenen Strukturen umzugehen. Zur Explizitmachung solcher Strukturen müssen meistens sehr aufwendige Verfahren eingesetzt werden, deren Erfolgsquote in der Regel nicht bei 100% liegt.

Ähnlich verhält es sich mit Textdateien. Einerseits gibt es hier gewisse Strukturen, die ein Computer erfassen kann (Worte, Zeilen und Absätze). Wenn es aber darum geht, die implizite Struktur von Texten zu erfassen, beispielsweise die Verfolgung des roten Fadens, so sind erneut nur sehr aufwendige Verfahren mit ungewisser Ausgabe einsetzbar.

Daten, die gar keine Struktur enthalten, gibt es nicht. Daten sind immer die maschinenlesbare Repräsentation von Informationen und Informationen stehen niemals für sich alleine. Per Definition besitzen sie immer eine Verknüpfung mit anderen Informationen, d. h. mit bereits vorhandenem Wissen. Unter unstrukturierten Daten werden in dieser Arbeit Daten verstanden, die keine explizite, für Computer geeignete Strukturbeschreibung besitzen.

Alles, was zwischen den beiden Extremen von unstrukturierten und strukturierten Daten liegt, ist semi-strukturiert. Semi-strukturierte Daten enthalten explizit Informationen über die eigene Struktur. Die Daten sind somit selbst beschreibend. Allerdings sind sie nicht streng typisiert und die Struktur kann (partiell) implizit sein [Bun97].

Der Begriff der semi-strukturierten Daten (SSD) wurde wesentlich durch das TSIMMIS-Projekt (The Stanford-IBM Manager of Multiple Information Sour-

ces, 1995-2000, u. a. J. Ullman, H. Garcia-Molina, J. Widom, Y. Papakonstantinou) geprägt. Dort ging es um die Integration heterogener Quellen, die durch so genannte Mediatoren gekapselt wurden. Zum Austausch der Daten wurde ein Datenformat für semi-strukturierte Daten verwendet.

Die Schätzungen der Marktforschungsunternehmen [BA03] gehen davon aus, dass 85-90% der weltweit vorhandenen Daten nicht voll strukturiert sind. Im Internet kommen etwa 200 Millionen neue Seiten jeden Monat hinzu. Die Verwaltung von nicht voll strukturierten Daten wurde daher als eines der wichtigsten ungelösten Probleme der IT-Industrie erkannt [BA03]. Jeder Büroangestellte benötigt momentan etwa ein Viertel seiner Zeit dazu, Informationen aus nicht strukturierten Datenbeständen wie dem Internet oder lokal vorhandenen Dokumenten zusammenzustellen. Häufig geschieht dies noch ohne den Einsatz von leistungsfähigen Hilfsmitteln. Die Dokumente im Intranet von Unternehmen sind teilweise noch nicht einmal in ein System zur Volltextsuche eingebunden.

Der potentielle Markt für Werkzeugunterstützung in diesem Gebiet ist Milliarden Euro schwer. Als Schlagwort hat sich der Begriff *Content Intelligence (CI)* ausgebildet. Dazu gehören die folgenden wichtige Bereiche [BA03]:

- Plattformen für Content-Applikationen
- Erweiterte Suchfunktionalität
- Entdeckung von Daten in nicht strukturierten Daten (*Data Mining*)
- Datenintegration mit anderen Systemen oder Alt-Systemen

Wenn es um so viel Geld und viele, verschiedenen Softwaresysteme geht, dann sind Standards auf den untersten Ebenen wichtig, um Interoperabilität zu gewährleisten. Hierum geht es im folgenden Abschnitt.

1.3 Die eXtensible Markup Language (XML)

Der Inbegriff für semi-strukturierte Daten lautet XML. An dieser Stelle soll ein erster, kurzer Überblick über diese Technologie gegeben werden, der in den folgenden Kapiteln noch vertieft wird.

Die *eXtensible Markup Language* ist noch sehr jung, dafür aber inzwischen in aller Munde. Durch XML wurde geradezu ein „Hype“ ausgelöst. Die Geschichte von XML begann im Sommer 1996, als die XML Working Group durch John Bosak (Sun Microsystems) gegründet wurde. Diese Gruppe sollte in Zusammenarbeit mit dem W3C einen Standard für semi-strukturierte Daten erarbeiten. Die Ziele bei der Entwicklung waren [May03]:

- Unterstützung einer großen Bandbreite von Anwendungen
- Für Menschen lesbar und nachvollziehbar
- Einfache Erstellung
- Einfache Verarbeitung von XML in Applikationen
- Über das Internet direkt und auf einfache Weise nutzbar (Unicode-Files)
- Kompatibel zu SGML [1]

Knapp zwei Jahre später, im Februar 1998, wurde eine Recommendation vom W3C für XML 1.0 [XML98a] herausgegeben. In XML 1.0 wird eine hierarchische Auszeichnungssprache beschrieben, die auf den Konzepten der *Markuptags* (im Folgenden *Knoten* genannt), der Attribute und weiteren Basiselementen aufbaut. Ein XML Dokument ist zunächst eine Baumstruktur von Knoten, auf denen eine Ordnung besteht. Durch spezielle Notationen kann der Baum eines XML Dokuments auch zu einem Graphen erweitert werden.

[1] Die Standard Generalized Markup Language (SGML), die 1986 zum ISO-Standard [ISO] wurde (ISO 8879), dient der Strukturierung und Auszeichnung von Dokumenten, und ist im Verlagswesen weit verbreitet. Auch HTML (1991) [HTM] ist eine SGML Applikation.

Inzwischen liegt die dritte Edition des XML 1.0 Standards vor. Die wesentliche Änderung betraf die Einführung von Namensräumen, um Kollisionen von Knotennamen bei komplexen Szenarien aus dem Weg zu gehen.

Im Februar 2004 wurde die erste Recommendation zu XML 1.1 [XML04] verabschiedet. Einige Erweiterungen des Unicode Standards hatten direkte Auswirkungen auf den XML Standard, was sich in XML 1.1 niederschlug. Für eine detaillierte Beschreibung von XML wird auf die angegebenen Quellen, sowie auf einige Tutorials und Bücher verwiesen [XML98b, Wal98, HM02].

Der XML Standard stammt ursprünglich aus dem Gebiet des Dokumentenmanagements und der Datenformate für das Internet. XML wird aber auch bei datenintensiven Anwendungen eingesetzt. Aus diesem Grund werden zwei Arten von XML unterschieden: *datenzentrisches XML* und *dokumentenzentrisches XML*. Die Unterscheidung ist allerdings erneut nicht eindeutig. Einige Kriterien für eine Klassifikation sind in Tabelle 1.1 zusammengestellt.

Datenzentrisches XML	**Dokumentenzentrisches XML**
• sehr regelmäßige Struktur mit 'Datenfeldern' • wenig Text • keine natürlich vorgegebene Baumstruktur	• Baumstruktur mit viel Text • unregelmäßige Elementstrukturen • logisches Markup des Dokuments • Annotationen des Textes durch Elemente/Attribute

Tabelle 1.1: Kriterien für die Unterscheidung zwischen daten- und dokumentenzentrischem XML

Neben dem eigentlichen XML Standard gibt es weitere darauf aufbauende Standards des W3C. Essentielle Standards gibt es beispielsweise in den Bereichen der XML Grammatiken (z. B. *XML Schema* [XSD]) und der XML Anfragesprachen (z. B. *XPath* [XPa99] und *XQuery* [XQu03a]).

XML Grammatiken dienen dazu, XML Dokumente auf erlaubte Dokumente (valide Dokumente) einzuschränken. Dadurch können domänenspezifische XML Sprachen definiert werden, die es mittlerweile in vielen verschiedenen Bereichen bereits gibt. Die Vielfalt der Sprachen und Anwendungen ist enorm. Es folgen einige wenige Beispiele hierzu:

- SVG zur Beschreibung von Vektorgrafiken
- GML für Geographische Informationsverarbeitung
- BPML (Business Process Modeling Language) im Bereich von Geschäftsprozessen
- XOL (*XML Ontology exchange Language*) im Bereich von Ontologien und des Semantic Webs
- chessML oder ChessGML zur Beschreibung von Schachpartien inklusive Kommentaren

Seit dem Erscheinen von XML Schema [XSD] ist es nicht mehr korrekt, XML mit semi-strukturierten Daten gleichzusetzen. Wenn zu einer Menge von XML Dokumenten ein XML Schema vorhanden ist, so kann das Schema die Daten komplett und typsicher beschreiben. XML Dokumente können also voll strukturierte Daten enthalten. Umgekehrt können wie oben erwähnt Datenbanksysteme auch unstrukturierte Daten speichern. Jede Technik bietet also Möglichkeiten an, die über Ihren ursprünglichen Einsatzhorizont hinausgehen.

Um in XML Daten schnell und einfach Informationen zu finden, gibt es u. a. die Anfragesprachen XPath und XQuery. XPath enthält die wesentlichen Konzepte, um gezielt Stellen in XML Dokumenten zu lokalisieren. XQuery ist von Ihrer Mächtigkeit her mit SQL vergleichbar. In den folgenden Kapiteln wird bei Bedarf genauer auf diese und weitere XML Standards eingegangen.

In realen Applikationen wird XML für viele verschiedene Zwecke eingesetzt. Immer, wenn mit semi-strukturierten Daten umgegangen wird, kommt XML als Speicherungsformat in Betracht. Wenn das Wissen einer Firma gespeichert

werden soll, kann dies sehr gut in XML erfolgen. Wissen lässt sich selten in feste relationale Schemata pressen, wodurch eine Verarbeitung in herkömmlichen Datenbanksystemen meistens ausscheidet.

Ebenfalls kommt XML dann in Betracht, wenn es schwierig oder unmöglich ist, Schemata für die Daten anzugeben, die verwaltet werden sollen. Im privaten Bereich ist beispielsweise der *Personal Information Store* von Interesse. Dieser enthält Daten, die einer Person wichtig sind. Hierfür kann es kein Schema geben, da der Benutzer völlig verschiedene Daten ablegen will, beispielsweise ein Küchenrezept, die Interessensgebiete von Freunden, Zitate, Textpassagen oder Telefonlisten.

XML spielt auch im Bereich annotierter Daten eine gewichtige Rolle. In der Bioinformatik will man zu DNA-Sequenzen textuell bestimmte Bedeutungen zuordnen. Sehr beliebt ist XML für die Speicherung von Konfigurationsdaten von Softwaresystemen. Früher wurden diese Daten i. d. R. in reinem Textformat abgelegt. Konfigurationsdaten sind allerdings häufig hierarchisch strukturiert. Dies liegt daran, dass Softwaresysteme aus Komponenten bestehen, die wiederum rekursiv aus kleineren abgeschlossenen Teilen bestehen. XML drängt sich hierfür geradezu als Speicherungsformat auf. Die Systemdatenbank von Windows, die so genannte *Registry* [Rus97, Rus99, Coo02] besitzt ebenfalls eine hierarchische Struktur, die mit XML gut modelliert werden könnte.

Eine weitere Rolle spielt XML beim Datenaustausch zwischen Applikationen. Zwei Applikationen können Datenaustausch betreiben, indem die Daten von der einen Applikation nach XML exportiert und von der anderen importiert werden. Eventuell ist hierbei noch ein Zwischenschritt notwendig, der eine mehr oder weniger aufwendige Konvertierung der XML Dateien durchführt. Solche Konvertierungen werden durch XML Standards wie z. B. *XSL Style Sheets* [XSL01] und dazugehörige Werkzeuge auf einer hohen Abstraktionsebene unterstützt.

Inzwischen benutzen viele Applikationen XML sogar als primäres Speicherungsformat (z. B. Microsoft Office 2003). Dadurch entfällt der Exportschritt. Allerdings können die erstellten XML Dateien komplexer sein, als dies für einen sinnvollen Export nötig wäre. Deshalb erscheint eine zusätzliche Exportfunktion für gewisse Dokumente nach wie vor sinnvoll zu sein.

Für Applikationsprogrammierer, die mit objektorientierten Sprachen arbeiten, existieren inzwischen Techniken, die eine einfache Definition einer Abbildung zwischen Objekten und einer XML Repräsentation erlauben. Für Java ist hier *JAXB (Java Architecture for XML Binding)* [JAX04a] der Standard. Von JAXB gibt es inzwischen eine ganze Reihe von guten Implementierungen. Im open-source Bereich ist hier beispielsweise Castor [Cas04] zu nennen.

Insgesamt lässt sich sagen, dass die XML Technologie eine Vielzahl von Möglichkeiten anbietet, auf die immer mehr Applikationen angewiesen sind. XML hat gleichzeitig eine Diskussion um standardisierte Datenformate innerhalb von größeren *Communities* ausgelöst. Durch die vielen, fest definierten XML Sprachen ist innerhalb von Communities ein Datenaustausch möglich geworden. In einigen Fällen gibt es allerdings auch XML Sprachen, die miteinander konkurrieren. Dies kann positive Effekte mit sich bringen, da hierdurch die vorhandenen Sprachen einer kritischen Prüfung unterzogen werden. Allerdings kann die Konkurrenz auch dazu führen, dass Systeme nicht mehr direkt interoperieren können und Konvertierungen zwischen verschiedenen Sprachen erfolgen müssen.

XML ist eine junge Technologie, deren Basis-Standards noch eine weitere Entwicklung durchlaufen werden. Bis heute haben sich die Standards als sehr brauchbar erwiesen, da im Laufe der Zeit keine nennenswerten Inkompatibilitäten in den Tools entstanden sind. Durch die fortschreitende Standardisierung von Techniken um XML herum werden in Zukunft Applikationsprogrammierern erweiterte Möglichkeiten zur Verfügung stehen, wodurch neue Anwendungsgebiete für XML basierte Systeme hinzukommen werden.

Durch die Vereinheitlichungen im XML Bereich und durch die open-source Bewegung werden künftig immer mehr anwendungsspezifische Softwarebibliotheken entstehen. Dies kann die Applikationsprogrammierung deutlich erleichtern und selbst einfachen Applikationen ganz neue Möglichkeiten eröffnen. Besonders datenintensive Anwendungen, deren Standards teilweise noch in der Entwicklung sind, werden in dieser Hinsicht in Zukunft profitieren.

1.4 XML und Datenbanken

XML entwickelt sich zunehmend in Richtung datenintensiver Anwendungen. Immer mehr semi-strukturierte Daten werden inzwischen langfristig in XML gespeichert. Hierfür ist die Unterstützung für semi-strukturierter Daten in herkömmlichen Datenbanksystemen bislang nur unzureichend.

In vielen Applikationen wären XML Datenbanksysteme sehr von Vorteil. Ein Beispiel hierfür sind Internetanwendungen, die HTML als Darstellungsformat produzieren müssen. Eine Konvertierung von XML nach HTML ist mit XSL Style Sheets problemlos möglich (siehe Abbildung 1.1). XSL kann auch zur Transformation von XML in andere XML Formate eingesetzt werden, beispielsweise bei der Transformation in das Grafikformat SVG. Dieses speichert Vektorgrafiken in XML Syntax und findet derzeit im Internet immer mehr Verbreitung.

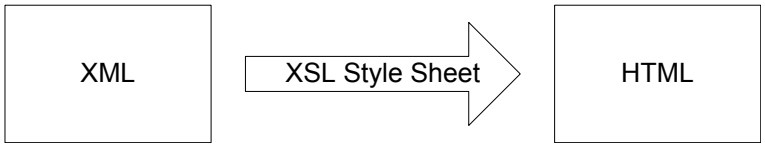


Abbildung 1.1: Die Verarbeitung von XML mittels XSL Style Sheets

Viele *Content Management Systeme (CMS)* speichern ihre Daten in XML Dokumenten. Ein CMS ist ansonsten von den benötigten Diensten her einer typischen Datenbankapplikation sehr ähnlich. In einem CMS existieren viele verschiedene Benutzer mit unterschiedlichen Berechtigungen. Benutzer können neue Daten (Internetseiten, Layoutinformationen, Navigationsstrukturen, etc.) einfügen, diese verändern oder löschen. Dabei muss das System mehrbenutzerfähig sein, und die Inhalte müssen performant auf einem Webserver der restlichen Welt zur Verfügung stehen. Weiterhin ist eine schnelle Suchfunktion wichtig, die ähnlich der einer Suchmaschine im Internet funktioniert.

Die meisten heutigen CMSe legen ihre XML Daten entweder als Textdateien, oder als CLOB in einer Datenbank ab. Beides hat zur Folge, dass die Änderung eines einzelnen Datums bereits einen hohen Aufwand mit sich bringt. Das System muss dazu die komplette XML Datei in den Hauptspeicher laden, eine

Speicherstruktur (beispielsweise einen DOM-Baum) aufbauen, die Änderung vornehmen und anschließend die komplette Datei wieder wegschreiben. Das Grundübel besteht darin, dass sowohl Textdateien als auch CLOBs die Struktur der Dokumente nicht berücksichtigen. Es ist also keine gezielte strukturelle Suche innerhalb der Dokumente, so wie sie auf dem Externspeicher liegen, möglich.

An dieser Stelle versuchen *native XML Datenbanksysteme* die vorhandenen Technologien um XML herum zu ergänzen. Sie bieten die Dienste von herkömmlichen Datenbanken an, also Mehrbenutzerfähigkeit, Recovery, Sicherheit usw. Darüber hinaus stellen sie *nativen XML Speicher* zur Verfügung. Dieser benutzt spezielle, ausschließlich zu diesem Zweck entwickelte Strukturen, die bei Verwaltung von XML Dokumenten mit wenigen Zugriffen auf den Externspeicher auskommen. Dabei wird der Struktur der XML Dokumente Rechnung getragen. Bei kleinen, lokal begrenzten Operationen innerhalb eines Dokuments muss beispielsweise nicht das komplette Dokument bearbeitet werden, sondern es reichen wenige, gezielte Externspeicherzugriffe dafür aus.

Für die Speicherung von XML gibt es in der Datenbankliteratur noch einen zweiten, durchaus gebräuchlichen Ansatz. Die Speicherungsschicht kann XML Dokumente auf Relationen abbilden, die dann auf einfache Art und Weise in herkömmlichen Datenbanksystemen abgelegt werden können. Bislang wurde die Leistung solcher Abbildungen jedoch noch nicht mit der Leistung nativer Strukturen verglichen.

Neben nativen XML Datenbanksystemen gibt es inzwischen spezielle XML Erweiterungen für herkömmliche (objekt-)relationale Datenbanksysteme. Bisweilen enthalten diese so genannten *XML befähigten Datenbanksysteme* keine native XML Speicherung, sondern nur Möglichkeiten, XML Dokumente als CLOB zu speichern und Anfragen in den XML Sprachen XPath oder XQuery auszuführen. Dadurch ist die Leistung bei Anfragen und Änderungsoperationen meistens nicht zufriedenstellend.

1.5 Schlussfolgerungen

Die Notwendigkeiten in der Applikationsprogrammierung erfordern ergänzend zu XML neue Datenbanksysteme für semi-strukturierte Daten. Dabei sind die aus herkömmlichen Datenbanksystemen bekannten Dienste wie Mehrbenutzerfähigkeit, Recovery und Sicherheit zur Verfügung zu stellen. Diese neuen Systeme müssen außerdem höhere XML Anfragesprachen wie XPath und XQuery beherrschen und bei XML Anfragen mit den Ressourcen Hauptspeicher, Externspeicher und Rechenzeit umgehen.

Somit ist die effiziente Speicherung von XML eine wichtige Basis für diese neue Art von Datenbanksystemen. Das Ziel der vorliegenden Arbeit ist es, grundlegende Fragen zur Speicherung von XML zu beantworten. Dazu wurde ein Framework innerhalb der Java Bibliothek XXL[2] [BDS00, BBD+01, XXL] entwickelt, mit dem sich verschiedene Ansätze fair miteinander vergleichen lassen. In dieses Framework wurden neue Ideen integriert, die sich in Experimenten als sehr gewinnbringend zeigten. Es wurden Indexstrukturen für nativen XML Speicher untersucht und ein neuartiger Signaturindex implementiert, der auf einem Konzept für Aggregate basiert.

Die jüngere Literatur hat verdeutlicht, dass für native XML Datenbanksysteme viele Komponenten heutiger (objekt-)relationaler Datenbanksysteme neu überdacht werden müssen [Kan03]. Genannt sei hier, dass für die effiziente Recovery im XML Datenbanksystem *Natix* aus Gründen der Performanz neue Wege eingeschlagen werden mussten.

Alles in allem können native XML Datenbanksysteme momentan noch nicht als gut erforscht gelten. Die kommerziellen Anbieter in diesem Bereich publizieren leider nur sehr wenig über die Interna ihrer Systeme. Die von ihnen herausgegebenen Publikationen genügen weiterhin nur selten wissenschaftlichen Ansprüchen. Die für XML Datenbanksysteme neu entwickelten Techniken fallen

[2] Die Bibliothek XXL wird von der Arbeitsgruppe Datenbanksysteme am Fachbereich Mathematik und Informatik der Philipps-Universität Marburg entwickelt. Sie stellt dem Applikationsprogrammierer Datenbankfunktionalität zur Lösung verschiedenster Aufgaben zur Verfügung. In XXL integriert sind u. a. Komponenten für die Datenspeicherung und eine allgemeine Operatoralgebra zur Unterstützung von Anfragesprachen. Die Beschreibung der für die Arbeit wichtigen Funktionalität von XXL ist Teil von Kapitel 3.

häufig auch unter Geheimhaltung, weil sie am Markt einen Vorteil gegenüber Konkurrenten bedeuten. Es gibt auf dem Gebiet der nativen XML Datenbanksysteme also noch viele Möglichkeiten, wissenschaftlich zu arbeiten und neue Prinzipien zu entdecken.

1.6 Aufbau der Arbeit

Im zweiten Kapitel werden zunächst einige für die Arbeit wesentliche Begriffe erwähnt und erläutert. Dazu gehört die Beschreibung des Modells des Externspeichers, die Vertiefung von XML und einige allgemeine Feststellungen zur Leistungsmessung.

In nativen XML Datenbanksystemen wird auf Basiskomponenten zurückgegriffen, die bereits aus (objekt-)relationalen Datenbanksystemen bekannt sind. Gleichzeitig müssen einige Basiskomponenten neu überdacht werden. Im dritten Kapitel erfolgt die detaillierte Beschreibung einiger wichtiger Basiskomponenten. Zunächst wird dazu ein Überblick über XXL gegeben, der an einigen für die Arbeit wichtigen Stellen vertieft wird.

Neben den aus der Literatur bekannten Verfahren werden auch einige neue bzw. optimierte Verfahren vorgestellt. In Evaluierungstests wird die Leistung der veränderten Verfahren eingeordnet. Gleichzeitig wird darauf eingegangen, wie sich der Austausch von Strategien innerhalb der Komponenten auf das Leistungsverhalten des Gesamtsystems auswirkt.

Im vierten Kapitel geht es um relationale XML Daten. Dies sind datenzentrische XML Dokumente, die auf einfache Art und Weise von XML in das relationale Modell und zurück konvertiert werden können. Solche Daten finden sich relativ häufig im Internet. Es existieren bereits unzählige Hilfsprogramme, die relativ komfortabel und flexibel solche XML Dokumente in relationale Datenbanken einlesen können. Für XXL stellt sich die Frage, wie solche Konvertierungen effizient und mit guter Unterstützung für Pipelining in Operatorbäumen umgesetzt werden können.

Das fünfte Kapitel stellt das zentrale Kapitel dieser Arbeit dar. Hier werden

native Speicherungsstrukturen vorgestellt, die nicht auf herkömmlichen relationalen Datenbanksystemen aufsetzen. Es werden verschiedene Strategien für die Speicherung von XML vorgestellt. Diese zeichnen sich beispielsweise durch gute Eigenschaften bei Änderungsoperationen aus. Im Evaluierungsteil werden die unterschiedlichen Strategien mittels des Benchmarks XMark [SWK+02] miteinander verglichen. Es folgt die Beschreibung von Problemen und deren Lösungsmöglichkeiten bei der Implementierung von Indexen auf nativem XML Speicher. Im Framework selbst wurde ein Signaturindex implementiert, der auf einem effizienten Konzept für Baumaggregate aufsetzt.

Das sechste Kapitel enthält Vergleichstests der nativen Speicherungsform aus Kapitel 5 mit Speicherungsverfahren für XML, die auf relationalen Datenbanksystemen aufsetzen. Aus der Literatur wurden drei solche Verfahren ausgesucht und im selben Framework wie der native Speicher implementiert. Dadurch können erstmals diese Speicherungsverfahren in derselben Umgebung mit nativen XML Speicherungsverfahren verglichen werden.

Anschließend wird im siebten Kapitel darauf eingegangen, dass XQuery nicht die einzig wichtige Variante einer XML Anfrageverarbeitung ist. Ströme von XML Dokumenten erfordern physische Operatoren, die komplette XML Dokumente auf neue XML Dokumente abbilden.

Das achte Kapitel enthält abschließend eine kurze Zusammenfassung der wichtigsten Ergebnisse der Arbeit, sowie die Diskussion von derzeit offenen Fragen. Im Ausblick wird auf eine mögliche Datenbankwelt von morgen eingegangen, die Elemente dieser Arbeit integriert und deren inhärenten Ideen konsequent umsetzt.

Der Anhang enthält noch einige Hinweise zum Quelltext.

Kapitel 2

Grundlagen

Systeme zur Verarbeitung von XML enthalten zwar viele neu entwickelte Komponenten, jedoch erfinden sie das Rad nicht komplett neu. Sie basieren auf einer Reihe von Komponenten, die bereits in existierenden Softwaresystemen eingesetzt werden.

In diesem Kapitel werden grundlegende Probleme im Umfeld großer, datenintensiver Softwaresysteme beschrieben. Hierbei wird der in der weiteren Arbeit eingeschlagene Lösungsweg motiviert. Anschließend erfolgt eine Vertiefung der Techniken aus dem XML Umfeld. Das Kapitel schließt mit einer Aufstellung der Daten der Testrechner, sowie Angaben zur verwendeten Notation.

2.1 Die Implementierungssprache

Die erste Design-Entscheidung betraf die Wahl der Implementierungssprache. Hier gibt es viele Aspekte, die gegeneinander abgewogen werden mussten. Aus verschiedenen Gründen wurde die Programmiersprache Java [Jav04] gewählt:

- Objektorientierung
- Mächtigkeit der Sprache
- Plattformunabhängigkeit
- Vorhandene Bibliotheken (SDK, XML Parser, XXL, etc.)

- Nutzung künftiger Bibliotheken, da viele Wissenschaftler im XML Bereich die Programmiersprache Java einsetzen

- Reduzierter Entwicklungsaufwand im Vergleich zu C++, da weniger fehleranfällig

Gegen Java spricht bei gewissen Teilaufgaben die Performanz. Anwendungen, die viele kleine Objekte erzeugen, lassen sich in C++ häufig etwas effizienter implementieren. Allerdings wurden die größten Leistungsprobleme mit der Einführung der *HotSpot Virtual Machine*, sowie der inkrementellen *Garbage Collection* beseitigt. Die meisten Java Anwendungen können inzwischen mit C++-Anwendungen mithalten [SH03a, SH03b]. Auch bei C++ selbst gibt es große Unterschiede in den Optimierungskünsten der verschiedenen Compiler. Dies kann durchaus mehr als 20% ausmachen [SS04].

Ein weiteres Problem von Java ist, dass es keine maschinennahe Programmierung erlaubt [SMFH01]. Wenn Daten beispielsweise von der Festplatte in Blöcken eingelesen werden, so ist anschließend eine explizite Konvertierung der Bytefelder in Java Objekte nötig. In C++ genügt das Lesen der Bytefelder gefolgt von einer Umwandlung des Zeigertyps. Dieses Verfahren ist allerdings kein guter Stil, da es weniger an eine Hochsprache, denn an Assembler erinnert. Außerdem ist die Technik fehleranfällig und macht weiterhin die implementierte Anwendung abhängig von der Plattform und sogar vom Compiler.

Über die Erfahrungen anderer Projekte aus dem Bereich datenintensiver Systeme wird im Zusammenhang mit dem Telegraph Projekt in [SMFH01] berichtet. Die Autoren sehen Vor- und Nachteile von Java. Allerdings hat sich seit diesem Bericht in der Entwicklung von Java viel getan, so dass viele der im Bericht genannten Nachteile für die aktuelle Java-Version nicht mehr gelten.

Bei all der Diskussion um die Vor- und Nacheile von Programmiersprachen darf jedoch nicht vergessen werden, dass es im Rahmen dieser Arbeit nicht darum geht, ein möglichst performantes System zu erstellen, sondern darum, gewisse Verfahren geeignet gegeneinander zu testen. Dies erfordert es nicht, die maximal mögliche Performanz auf einem vorgegebenen Rechner zu erreichen, sondern die Leistung zweier Verfahren relativ zueinander fair zu bewerten.

2.2 Leistungsbewertung

Seitdem es Computer gibt, ist die faire Leistungsbewertung von Hard- und Software problematisch. Eine Messung der Leistung kann beispielsweise anhand von theoretischen Modellen erfolgen. In die ermittelten Leistungswerte fließen hierbei z. B. die Taktfrequenz des Rechners, sowie Art und Anzahl der ausgeführten CPU-Operationen ein. Spätestens seit RISC mit seinen langen Pipelines innerhalb der Prozessoren und Merkmalen wie der Sprungvorhersage ist diese Art der theoretischen Leistungsmessung jedoch fragwürdig, da komplexe Vorgänge häufig nicht in einfachen Formeln berücksichtigt werden können.

Somit ist in vielen Fällen nur die praktische Messung der Laufzeit für die Bestimmung der Leistung adäquat. Solche Messungen erfolgen anhand einer in einem Benchmark vordefinierten Last. Dadurch lässt sich jedoch gleich folgern, dass die Übertragung der Ergebnisse eines Benchmarks auf eigene Szenarien schwierig ist. Benchmarks spiegeln die Leistung nur innerhalb eines gewissen, speziellen Einsatzbereiches wieder. Bei der Übertragung auf eigene Szenarien muss zunächst sehr genau geprüft werden, in wie weit das eigene Szenario dem Szenario des Benchmarks entspricht.

Zusätzlich gehen in die Ergebnisse eines Benchmarks bereits Faktoren ein, die unbefriedigend sind. Compilerhersteller optimieren ihre Produkte nach den selben Benchmarks, mit denen anschließend die Leistung des Compilers angepriesen wird. Somit wird durch einen Benchmark ein Leistungswert bestimmt, der ganz entscheidend die Fähigkeit bewertet, wie gut ein Compiler den Quelltext des Benchmarks optimiert.

Inzwischen werden Benchmarks allerdings von Anfang an so konzipiert, dass es den Compilerherstellern schwer gemacht wird, speziell zu optimieren. Dies wird zum einen durch die reine Größe des Benchmarks erreicht und zum anderen dadurch, dass die Optimierung einzelner Operationen nur einen kleinen Einfluss auf das Messergebnis hat. Somit sind Fälle wie früher beim Dhrystone-Benchmark [Wei84] ausgeschlossen, wo eine einzelne Compileroptimierung die Ergebnisse um Faktoren verbessert hat. Damals konnte aus den Ergebnissen des Benchmarks mit einer guten Trefferquote bestimmt werden, ob der verwendete Compiler die entsprechende Optimierung gemacht hatte oder nicht.

Aus diesen Ausführungen lässt sich weiterhin schließen, dass niemals Algorithmen miteinander verglichen werden dürfen, die nur in Binärcode vorliegen. Hier ist die Gefahr zu groß, dass nur die Eigenschaften der jeweils benutzten Compiler getestet werden. Somit lässt sich sagen, dass die zu vergleichenden Verfahren immer im selben Framework implementiert werden müssen. Doch dies alleine ist nicht ausreichend. Realistisch betrachtet haben die Fertigkeiten des Programmierers ebenfalls einen großen Einfluss auf die Laufzeit. Programmierer schreiben nahezu niemals einen fehlerfreien und noch seltener einen optimalen Quelltext. Als perfekt können eventuell noch kurze und verifizierte Programme gelten, allerdings niemals große Softwaresysteme.

Trotzdem sollen Leistungsbewertungen durchgeführt werden. Dies kann nur dann erfolgen, wenn die Qualität des Quelltextes hoch ist. Eine Bewertung der Qualität von Quelltext ist jedoch schwierig und liegt nicht im Themenbereich dieser Arbeit. Es sollen aber dennoch einige Vorgehensweisen genannt werden, mit denen die Qualität der im Rahmen dieser Arbeit erstellten Quelltexte gesteigert wurde (Fehlerbeseitigung, Performanz). Zu nennen sind hierbei:

- Die Quelltexte der verwendeten Systemen (Java), Bibliotheken (XXL) und der Arbeit selbst sind *öffentlich*. Somit kann weltweit jeder Fachmann Kritik üben, die zu einer Verbesserung des Quelltextes führen kann.

- Für die meisten Komponenten von XXL und der in dieser Arbeit neu entwickelten Komponenten gibt es *Testfälle*, die dazu dienten, dass viele Fehler bereits frühzeitig während der Entwicklung ausgemerzt werden konnten. Fehler in Programmen können dazu führen, dass die Berechnung des eventuell trotzdem korrekten Endergebnisses aufwendiger wird. Die Testfälle wurden immer parallel zu der Entwicklung der jeweiligen Komponenten geschrieben. Sie erfüllen weiterhin den Zweck, dass die Korrektheit von künftigen Änderungen ohne viel Aufwand getestet werden kann.

- Mittels eines so genannten Profilers können *Hot Spots* erkannt werden. Dies sind Methoden im Quelltext, die bei der Ausführung die meiste Zeit benötigen. Wenn man annimmt, dass eine Optimierung nur lokal wirkt

und die Laufzeit der Methode verbessert, so ist eine einfache Beispielrechnung möglich: Wenn alle Hot Spots unter 10% der Gesamtlaufzeit benötigen, so bringt die beste mögliche Einzeloptimierung maximal eine Laufzeitverbesserung von 10%.

Für alle grundlegenden Verfahren dieser Arbeit wurden jeweils die ersten 10-20 Hot Spots untersucht und optimiert. Die Schlüsse, die aus der Existenz der Hot Spots gezogen wurden, konnten auch dazu führen, globale Optimierungen durchzuführen, die prinzipiell völlig neue Hot Spots erzeugen können. In diesem Fall wurden die ersten 10-20 Hot Spots wieder aufs neue untersucht.

All dies bedeutet selbstverständlich nicht, dass der nicht speziell optimierte Quelltext langsam ist. Allerdings wurde er nur einer genauen Überprüfung unterzogen, weil dessen Einfluss auf die Gesamtlaufzeit weit unterhalb der Messungenauigkeit liegt.

Bei der Leistungsmessung in Java kommen noch einige weitere beachtenswerte Punkte hinzu. Java verwendet eine virtuelle Maschine [Sun02], welche die Klassendateien ausführt. Auch diese Maschine sucht nach Hot Spots im Code und optimiert diese und nur diese. Eine Optimierung des gesamten Codes wäre viel zu aufwendig. Dann würde zu viel Zeit für die Optimierung verbraucht und die Gesamtlaufzeit wieder ansteigen.

Die JVM lässt sich in zwei Modi betreiben, dem Client-Modus und dem Server-Modus. Im Server-Modus wird eine typische Last angenommen, wie sie auf einem Server vorliegt. Serverprozesse laufen i. d. R. sehr viel länger als Anwendungsprogramme auf Clients. Dadurch lohnen sich für die virtuelle Maschine weitergehende Optimierungen bei der Umsetzung der plattformübergreifenden Klassendateien auf die real vorhandene Serverhardware. Genau dies wird im Server-Modus gemacht.

Die im Rahmen dieser Arbeit entwickelte Software wird normalerweise auf einem Server eingesetzt und daher im Server-Modus betrieben. Wenn sie auf einem Client eingesetzt würde, so wäre die erzeugte Last jedoch so hoch, dass sich auch hier der Server-Modus lohnen würde.

Ein kritischer Punkt bei Performanzbetrachtungen in Java ist die so genannte

Garbage Collection. Java bereinigt den Speicher, indem es nicht mehr erreichbare Objekte selbstständig und automatisch entsorgt. Die Techniken für Garbage Collection haben sich mit den letzten Versionen von Java stark verbessert. Inzwischen läuft die Garbage Collection in eigenen Threads ab, so dass die Rechenzeit genutzt werden kann, in denen die eigentlichen Arbeiter-Threads durch E/A Zugriffe oder anderes blockiert sind. Weiterhin läuft die Garbage Collection in neueren Versionen inkrementell ab, so dass Applikationen nicht wie früher üblich für eine lange Zeit unterbrochen werden.

Laufzeittests in Java besitzen ein weiteres Problem. Dadurch, dass prinzipiell immer dynamisch gelinkt wird, werden Klassen erst bei Bedarf nachgeladen. Dies kann bei großen Systemen mit mehreren hundert benutzten Klassen durchaus einen Einfluss auf die Messergebnisse haben. Die Zeit für das Laden und Initialisieren der Java-Klassen ist bei einer Server-Anwendung wie einem Datenbanksystem allerdings nicht von Interesse, weil die meisten Klassen, die zum Basissystem gehören und in dieser Arbeit betrachtet werden, sowieso immer verfügbar sein müssen. Das einmalige Laden der Klassen beim Start des Serverprogramms fällt somit nicht ins Gewicht. Um Einflüsse durch das Klassenladen zu verhindern, ist ein Warmlaufen sinnvoll. Hierbei wird vor dem Testlauf versucht, mit möglichst wenig Aufwand auf eine Reihe von Klassen zuzugreifen, wodurch der Java Klassenlader sie lädt.

Generell ergeben sich in Java durch Hot Spot Compiler, Garbage Collection und das Nachladen von Klassen höhere Messungenauigkeiten als in C++. Wie sich später zeigen wird, sind diese Ungenauigkeiten jedoch akzeptabel für unsere Zwecke.

2.3 Das Modell des Externspeichers

Wesentlich für Datenbanksysteme ist ein effizienter Umgang mit dem Externspeicher. Jeder neue Datensatz muss, bevor er einer Applikation als eingefügt gemeldet wird, auf ein nicht flüchtiges Externspeichermedium geschrieben worden sein. Dies führt bei vielen datenintensiven Applikationen zu einem enormen Zeitaufwand.

Der folgende Abschnitt beschreibt nun die generellen Probleme beim Zugriff

auf den Externspeicher, wozu auch die Messung der Leistung von Externspeicherverfahren zählt.

Computersysteme arbeiten heutzutage mit einer Hierarchie von Speichersystemen. Auf dem Prozessor befinden sich die ersten beiden Ebenen, die Register und der Cache-Speicher. Diese beide Arten von Speicher sind sehr schnell zugreifbar, dafür allerdings auch sehr klein. Auf der dritten Ebene gibt es den Hauptspeicher des Rechners, der deutlich größer, aber auch um einiges langsamer ist. Diese drei ersten Ebenen der Speicherhierarchie sind flüchtig, d. h. nach dem Ausschalten des Rechners geht der Inhalt unwiederbringlich verloren. Dies ist beim noch größeren und noch langsameren Externspeicher nicht der Fall. Hier werden die Daten dauerhaft gespeichert.

Bei jedem Algorithmus muss nun versucht werden, dass die schnellen Speicher möglichst gut ausgelastet sind und dass möglichst wenige Zugriffe auf den nächst langsameren Speicher erfolgen. Im Rahmen dieser Arbeit spielt der Übergang zwischen Hauptspeicher und Externspeicher die entscheidende Rolle. Effekte durch eine verbesserte Nutzung der Prozessorregister können sowieso nicht mit einer höheren Programmiersprache untersucht werden. Effekte durch den Prozessorcache wurden in dieser Arbeit nicht betrachtet, da ihr Einfluss im Vergleich zum teuren Externspeicherzugriff bei den implementierten Algorithmen zu gering war.

Momentan ist das Externspeichermedium der Wahl die *Festplatte*. Sie enthält synchron rotierende Magnetscheiben, auf deren Oberflächen Informationen durch Schreib-/Leseköpfe magnetisch geschrieben bzw. gelesen werden. Eine schematische Skizze einer Festplatte ist in Abbildung 2.1 zu finden. Für jede Seite einer Oberfläche existiert ein Schreib-/Lesekopf[1]. Die Köpfe können durch einen Schrittmotor bewegt werden, und zwar immer nur alle gemeinsam. Durch diese Bewegung ist eine Positionierung möglich, die den Abstand der Köpfe vom Drehzentrum verändert. Man spricht hier von der Positionierung auf einem Zylinder. Ein Zylinder enthält auf jeder Oberflächenseite eine so genannte Spur. Auf den Spuren sind so genannte Sektoren angeordnet, die die

[1] In der Skizze wurde aus Gründen der Übersichtlichkeit nur ein Schreib-/Lesekopf pro Oberfläche eingezeichnet

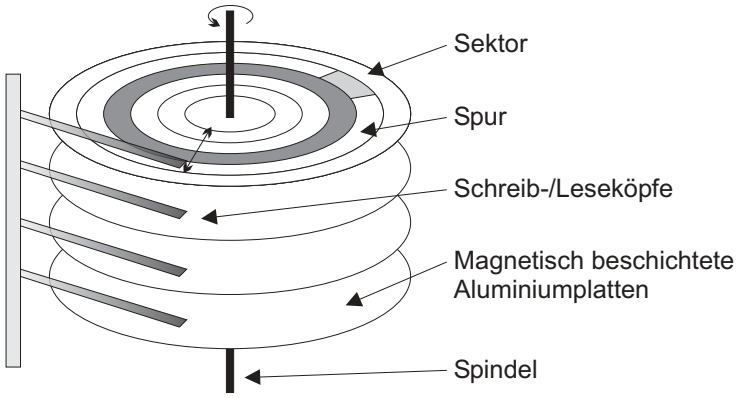


Abbildung 2.1: Schematischer Aufbau einer Festplatte

kleinste zugreifbare Speichereinheit auf einer Festplatte darstellen. Die Position eines Sektors kann durch eine dreidimensionale Adressierung mit Zylindernummer, Kopfnummer und einer Sektornummer eindeutig angegeben werden. Ein Sektor ist bei aktuellen Festplatten normaler Weise 512 Bytes lang.

In den letzten Jahren hat sich die Kapazität von Festplatten drastisch erhöht. Waren vor 20 Jahren noch $20MB$ das Maß aller Dinge, so sind es heute $200GB$ und mehr. Die Leistung der Hauptprozessoren von Personalcomputern stieg mit einer ähnlichen Geschwindigkeit. Allerdings konnte bei diesem Wettrennen um ein Mehr an Leistung die mittlere Zugriffszeit von Festplatten nicht mithalten ($50ms$ vor etwa 20 Jahren, heute etwa $5ms$). Das Verhältnis zwischen der benötigten Zeit für eine Prozessoroperation und der Zeit für einen Externspeicherzugriff liegt deshalb mittlerweile bei eins zu zehn Millionen. Wenn auf modernen Rechnern große Datenmengen bearbeitet werden, ist der Externspeicherzugriff ganz häufig der beschränkende Leistungsfaktor.

In der Forschung werden zwar regelmäßig neue Konzepte für den Externspeicher vorgestellt [EBC+03], bislang hatte dies jedoch nur einen geringen Einfluss auf die technische Entwicklung. Rein elektronische Speicherungsverfahren, wie *solid state disks (SSD)* [SSD04], sind zwar extrem schnell im Zugriff, jedoch auch sehr teuer, so dass sie sich bislang nicht in der Praxis durchsetzen konnten. In Zukunft können Änderungen der physikalischen Struktur von Externspeichern jedoch tiefgreifende Auswirkungen auf die Basiselemente von Datenbanksystemen haben [YAA03].

Der Zugriff auf Festplatten und andere Speichermedien erfolgt über das Betriebssystem. Dieses stellt Grundoperationen für Eingabe und Ausgabe (E/A) zur Verfügung: **open**, **read**, **write**, **seek** und **close**. Grundlegend ist, dass der Zugriff auf den Externspeicher immer nur in gewissen kleinsten Einheiten (Blöcke, Sektoren) erfolgt. Die Zeit, die für das Lesen eines Blocks von einer Festplatte benötigt wird, setzt sich zusammen aus einer *Suchzeit* t_s (Positionierung des Kopfs über der richtigen Spur), der *Rotationsverzögerung* t_r (Wartezeit, bis der gewünschte Sektor vorbeikommt) und der eigentlichen *Transferzeit* t_t. Bei *wahlfreien Zugriffen*, d. h. bei Zugriffen, bei denen der aktuelle Festplattenstatus nicht ausgenutzt wird, gilt [HR01]:

$$t_{wahlfrei} = t_s + t_r + t_t$$

Hierbei ist t_r die Rotationsverzögerung, die eintritt, sobald die Schreib-/Leseköpfe den gewünschten Zylinder erreicht haben. Weiterhin tritt noch eine Verzögerungszeit t_k für das Umschalten auf einen konkreten Schreib-/Lesekopf auf, die bei wahlfreien Zugriffen jedoch parallel zu t_s und t_r anfällt und daher nicht ins Gewicht fällt.

Die Verzögerungen, die zusätzlich durch Schnittstellenkarten entstehen, sind nicht nennenswert und können daher gegenüber den mechanischen Zeiten in den meisten Fällen vernachlässigt werden. Die Transferzeit t_t für einen Block ist abhängig von der Blockgröße B und der Transferrate T: $t_t = \frac{B}{T}$. Die Größe T hängt entscheidend von der Umdrehungszahl und den exakten Geometriedaten der Festplatte ab.

Bei modernen Festplatten kann im Mittel in etwa von den folgenden Werten ausgegangen werden: $T = 50MB/s, B = 512Bytes \Rightarrow t_t \approx 10\mu s, t_s = 8ms, t_r = 3ms \Rightarrow t_{wahlfrei} \approx 9,51ms$.

Die Transferzeit kann also guten Gewissens vernachlässigt werden. Bei sequentiellen Zugriffen, bei denen der nächste logische Sektor auf der Festplatte gelesen wird, entfällt die Rotationsverzögerung. Wenn sich der Sektor auf der selben Spur befindet, so entfällt auch die Suchzeit. Steht der gewünschte Sektor nicht auf der selben Spur, so darf die Kopfumschaltzeit nicht vergessen werden. Liegt

der Sektor sogar auf einem anderen Zylinder, so benötigt die Neupositionierung des Kopfes auf dem nächsten Zylinder etwa $1ms$.

Somit dauert ein sequentieller Zugriff im besten Fall nur etwa $10\mu s$, was fast 1000 mal schneller ist als ein wahlfreier Zugriff. Bei einem Wechsel auf den nächsten Zylinder sinkt der Vorteil auf einen Faktor von $\approx 9,4$. Zugriffsoperationen, die eine Bewegung des Plattenkopfes bewirken (im Folgenden mit *Seek* bezeichnet), müssen daher grundsätzlich vermieden werden.

2.3.1 Sequentiell ist nicht gleich sequentiell

Das sequentielle Lesen einer Datei durch eine Applikation kann allerdings trotzdem zu wahlfreien Zugriffen führen. Das liegt zum einen daran, dass nicht alle Daten der Datei auf einer Spur liegen müssen (*Kapazitäts Seek*), und zum anderen daran, wie die Datei geschrieben wurde (*Schreibstrategie Seek*) [Sch99]. Wenn beispielsweise vier Dateien zum Schreiben geöffnet sind und jede Datei nach und nach Daten erhält (wie beispielsweise bei einem externen Hash-Join), so liegt es an der Strategie des Betriebssystems, welche Sektoren auf der Festplatte an welche Datei vergeben werden.

Vergibt das Betriebssystem relativ große Blöcke[2] als kleinste Einheit, so müssen beim Schreibvorgang Seeks einkalkuliert werden. Die vom Betriebssystem den vier Dateien zugewiesenen Blöcke liegen dann nämlich eventuell nicht mehr auf dem selben Zylinder. Dafür stehen die Dateien nach dem Schreibvorgang in großen Teilen sequentiell auf der Festplatte, und somit kann ein anschließender Lesevorgang ohne viele Seeks erfolgen. Wenn das Betriebssystem kleinere Blöcke vergibt, so wird der Schreibvorgang der vier Dateien effizienter. Allerdings treten Seeks beim Lesen auf. Weiterhin liegen die Daten in diesem Fall näher zusammen (mit weniger freiem Speicher dazwischen), so dass Vorteile entstehen können, wenn ein Algorithmus auf mehrere der Dateien gleichzeitig zugreift.

Für Anwendungen, die ihre Daten nur einmal Schreiben und häufig lesen, ist die erste Variante günstiger. Bei speziellen Externspeicheralgorithmen wie beispielsweise einem Hash-Join kann jedoch die zweite Variante effizienter sein.

[2] Ein Block besteht aus einer Reihe von Sektoren, die auf einer Spur liegen.

Bei den obigen Ausführungen wurde davon ausgegangen, dass die Festplatte in den betrachteten Bereichen leer ist. Dies ist jedoch eine Annahme, die nicht generell gemacht werden kann. Für die Laufzeit eines Verfahrens spielt auch der Füllgrad der Platte eine wichtige Rolle. Ein hoher Füllgrad führt dazu, dass der freie Platz auf der Platte nicht an einem Stück vorliegt, sondern durch belegte Sektoren unterbrochen ist. Bei einem Füllgrad der Festplatte von über 80% kann die Leistung des Gesamtsystems deutlich leiden.

2.3.2 Leistungsbewertung

In dieser Arbeit werden Verfahren vorgestellt, die den Externspeicher sehr exzessiv nutzen. Somit ist es sehr wichtig, dass Externspeicherzugriffe korrekt und fair in die Leistungsbewertung eingehen. Wenn ein Verfahren den Externspeicher über das Dateisystem des Betriebssystems nutzt, so kann es bei den heutigen Schnittstellen leider niemals vorher entscheiden, ob ein Externspeicherzugriff einen Seek produziert oder nicht.

Weiterhin werden alle Zugriffe zunächst vom Betriebssystem gepuffert, was einen großen Einfluss haben kann. Beim Test eines neu entwickelten Verfahrens für ein Datenbanksystem werden die Puffer, die vom Verfahren selbst genutzt werden, normalerweise in der Größe relativ moderat gewählt. Das Verfahren soll schließlich hinterher auf einem Server laufen, auf dem noch hundert oder mehr ähnliche Verfahren gleichzeitig arbeiten. Hierdurch bleibt für das einzelne Verfahren effektiv weniger Puffer übrig. Bei Laufzeittests wird das Verfahren allerdings auf einem Rechner evaluiert, der ihm exklusiv zur Verfügung steht. Somit hat das Betriebssystem dieses Rechners extrem viel Hauptspeicher frei, das es für eine zusätzliche Pufferung verwenden kann. Es besteht also die Gefahr, dass nur noch die Leistung des Betriebssystempuffers gemessen wird und nicht die wirkliche Effektivität der Externspeicherzugriffe des neuen Verfahrens.

Daher wurde im Rahmen dieser Arbeit ein Framework erstellt, das eine komplette Kontrolle der Zugriffe auf die Festplatte ermöglicht. Dieses Framework wird in Kapitel 3.2.2.1 genauer vorgestellt. Hierbei konnte erreicht werden, dass sowohl Betriebssystempuffer, als auch Caches auf der Festplatte deaktiviert wurden.

Neben der experimentellen Bestimmung der Leistung von Algorithmen können auch theoretische Betrachtungen angestellt werden. Bei externspeicherlastigen Algorithmen, also solchen, die viel Zeit mit dem Warten auf Daten von der Festplatte verbringen, reicht es aus, einfach die Anzahl der Zugriffe zu zählen. Die Anzahl der Zugriffe ist dann in etwa proportional zu der tatsächlichen Laufzeit. Dieses Modell wird dadurch verbessert, indem man zwischen sequentiellen und wahlfreien E/A-Operationen unterscheidet. Sequentielle E/A-Operationen werden dann mit einem Faktor (beispielsweise 1/10 oder 1/20) gewichtet.

Eine weitere Verbesserung ist die Einführung einer dritten Klasse, nämlich der Klasse der *Mehrfachzugriffe*, bei denen der selbe Sektor zeitlich direkt aufeinanderfolgend zwei Mal angefragt wird. Diese Zugriffe treten bei schlechter Pufferung oder bei Write-Through Puffern auf. Sie sind teilweise noch schlimmer zu bewerten, als wahlfreie E/As. Es muss eine komplette Umdrehung der Festplatte abgewartet werden. Dadurch kann für diese Art Zugriff in theoretischen Modellen allerdings auch sehr genau angegeben werden, welchen Einfluss sie auf die Gesamtlaufzeit eines Verfahrens haben.

2.4 XML

Im Bereich um XML herum gibt es eine Reihe von wichtigen Standards. An dieser Stelle sollen zum besseren Verständnis der Arbeit einige bereits angesprochene XML Standards vertieft und weitere Standards eingeführt werden.

Einige dieser Standards stammen ursprünglich aus dem Gebiet des Dokumentenmanagements. Diese Orientierung ist bereits in der ursprünglichen Form der XML Grammatik zu erkennen. Die *Document Type Definition (DTD)* erlaubt keine komplette Typisierung eines Dokuments bis in alle Einzelteile (siehe Kapitel 1.3). Andere, besonders neuere Standards im XML Bereich sind dagegen eindeutig dem Datenbankbereich zuzuordnen.

2.4.1 Applikationsanbindung

Für Applikationen sind standardisierte Schnittstellen zwischen Programmiersprache und XML wichtig. Hierunter fallen Standards für Parser und Speichermodelle. Das *Document Object Model* ***(DOM)*** [DOM] ist ein Modell, das in Applikationen sehr einfach eingesetzt werden kann. Es baut den XML Baum im Hauptspeicher auf, wobei beispielsweise jeder Knoten in ein Objekt der Klasse `Element` umgesetzt wird. Applikationen können dadurch beliebig im Baum navigieren und benötigte Informationen ermitteln. Gleichzeitig ist auch das Einfügen, Ändern oder Löschen auf einfache Weise möglich. Der Nachteil von DOM ist, dass die Hauptspeicherrepräsentation i. d. R. die vielfache Größe des Ausgangsdokuments besitzt. Wirklich große XML Dokumente, deren interne Repräsentation nicht in den vorhandenen Hauptspeicher passt, lassen sich daher überhaupt nicht mit DOM bearbeiten.

Ein radikal entgegengesetzter Ansatz zu DOM ist die *Simple API for XML* ***(SAX)*** [SAX]. SAX macht einen gleichzeitigen Prä- und Postorder Durchlauf durch ein Dokument und meldet bei jedem Erreichen und Verlassen eines Knotens ein Ereignis an die Applikation weiter (an den *SAX Eventhandler*). Die Applikation muss sich daraufhin die für sie wichtigen Informationen eigenständig merken. Das Einfügen, Löschen und Ändern der Daten ist in SAX gar nicht vorgesehen. Der Vorteil von SAX ist, dass wenig Hauptspeicher verbraucht wird und auch keine lang andauernde initiale Phase beim Parsen nötig ist. Doch für Applikationen ist das Entwickeln der SAX Eventhandler ein mitunter sehr aufwendiger Prozess, der ohne Werkzeugunterstützung fehleranfällig ist.

2.4.2 Anfragesprachen

Einen wichtigen Baustein zur Gesamtarchitektur von XML stellen Anfragesprachen dar. In Applikationen müssen dadurch Anfragen an XML Dokumente nicht ausprogrammiert werden. Anfragen werden an einen Anfrageprozessor übergeben, der die Antworten berechnet und zurückgibt. Dieses Vorgehen ist gegenüber der Alternative des Ausprogrammierens sehr viel komfortabler, meistens effizienter und außerdem sehr viel weniger anfällig für Fehler. Schon bei relationalen Datenbanksystemen hat sich dieser Ansatz sehr bewährt.

Aufgrund der Herkunft von XML aus dem Dokumentenmanagement bieten sich außerdem noch Techniken des *Information Retrievals (IR)* [Fuh04] für die Anfrageverarbeitung an. Mittlerweile existieren einige Vorschläge, wie Techniken aus dem IR auf semi-strukturierte Daten angepasst werden können. In der Arbeit spielen diese Techniken jedoch keine Rolle.

2.4.2.1 XPath

Die grundlegende Anfragesprache für XML ist *XPath* [XPa99]. Mit dieser Sprache ist es möglich, eine Menge von Knoten in einem Dokument zu selektieren. Die Anwendung kann diese Knoten dann schrittweise traversieren.

XPath-Ausdrücke starten an einem gegebenen Knoten. Meistens ist dies die Wurzel eines Dokuments (absolute XPath-Ausdrücke), allerdings kann auch an einem vordefinierten Knoten gestartet werden (relative XPath-Ausdrücke). In einem XPath-Ausdruck kann dann entlang verschiedener Achsen im Dokument navigiert werden. Eine Übersicht über die in XPath definierten Achsen ist in Tabelle 2.1 zu finden.

Es existieren abkürzende Schreibweisen, die i. d. R. auch Verwendung finden. Anstatt `/child::name` kann `/name` geschrieben werden und anstatt `//descendant-or-self::name` ist das kürzere `//name` vorzuziehen. Einige Beispiele für XPath 1.0 sind in Tabelle 2.2 zu finden.

Neben der Navigation können auf jeder Ebene eines XPath-Ausdrucks ein oder mehrere Prädikate in eckigen Klammern angegeben werden. Wenn auf einer Ebene mehrere Prädikate vorhanden sind, so werden diese von links nach rechts ausgewertet, wobei das n-te Prädikat nur noch auf Stellen des Dokuments angewendet wird, die die vorherigen $n-1$ Prädikate erfüllen.

In einem Prädikat können relative oder absolute XPath-Anfragen als Unteranfragen verwendet werden. Relative Anfragen beziehen sich dabei auf die aktuelle Position im Dokument. Wenn die Auswertung einer solchen Unteranfrage eine oder mehrere Stellen im Dokument selektiert, so wird die Unteranfrage im Prädikat zu `true` ausgewertet. Beispielsweise kann eine Unteranfrage dazu genutzt werden, Elemente zu selektieren, die ein Kindelement mit einem speziellen Na-

Achse	Beschreibung
child („/“)	Navigation zu einem direkten Kindelement des aktuellen Knotens.
descendant	Navigation zu einem indirekten Kindelement des aktuellen Knotens.
parent	Direkter Vaterknoten
ancestor	Indirekter Vaterknoten
following-sibling	Nächster Geschwisterknoten
preceding-sibling	Vorhergehender Geschwisterknoten
following	Knoten, die in der Dokumentreihenfolge hinter dem aktuellen Knoten folgen.
preceding	Knoten, die in der Dokumentreihenfolge vor dem aktuellen Knoten stehen.
attribute	Attribute des aktuellen Knotens
namespace	Enthält die Namensraum Knoten des aktuellen Knotens.
self	Enthält nur den aktuellen Knoten
descendant-or-self („//“)	Enthält alle Knoten aus den Achsen descendant und self.
ancestor-or-self	Enthält alle Knoten aus den Achsen ancestor und self.

Tabelle 2.1: Die Achsen in XPath 1.0

XPath	Beschreibung
/child::book	Geht von der Wurzel zu den Kindelementen, die „book“ benannt sind.
/child::*	Geht von der Wurzel zu allen Kindelementen.
/child::*/child::book	Geht von der Wurzel zu allen Kindelementen und von dort zu allen direkten Kindelementen, die „book“ benannt sind.
/descendant-or-self::book	Geht von der Wurzel zu allen Elementen, die „book“ benannt sind.

Tabelle 2.2: Einige Beispiele in XPath 1.0

men besitzen. Die Anfrage „/book[editor/name]“ selektiert beispielsweise alle Bücher, unter denen ein Herausgeber mit Namen zu finden ist.

Weiterhin bietet XPath eine Reihe von eingebauten Funktionen, die in Prädikaten benutzt werden können, so z. B. *position()* und *last()*. Mit diesen Möglichkeiten lassen sich die folgenden Beispielanfragen formulieren: Finde das dritte bzw. das letzte book-Element unterhalb der Wurzel des Dokuments:

/book[position()=3]
/book[position()=last()]

Mit XPath lassen sich viele interessante Anfragen an XML Dokumente stellen. Die zweite Version von XPath wird demnächst noch erweiterte Möglichkeiten der Navigation bieten [XPa03]. Jedoch genügt auch XPath Version 2 noch nicht allen Ansprüchen an eine Anfragesprache für XML.

2.4.2.2 XQuery

XPath ist von der Mächtigkeit her nicht vergleichbar mit einer Sprache wie SQL in (objekt-)relationalen Datenbanksystemen. Verbundoperationen (Joins) zwischen Mengen selektierter Knoten sind in XPath 1.0 nicht möglich. Für diese Zwecke wird momentan die Anfragesprache *XQuery* [XQu03a] vom W3C entwickelt. XQuery besitzt momentan den *Last Call Working Draft* Status, d. h. es ist demnächst mit einer ersten *Proposed Recommendation* zu rechnen.

XQuery ist eine deklarative Anfragesprache, die auf einem strengen Typsystem aufbaut. XQuery besitzt einen Kern, auf den alle erweiterten Möglichkeiten der Sprache abgebildet werden können. Wie sonst bei funktionalen Sprachen üblich, existiert auch für XQuery die Beschreibung der formalen Semantik [XQu04].

XPath 1.0 ist eine Teilmenge von XQuery, so dass in den Klauseln von XQuery Ausdrücke in XPath Syntax verwendet werden können. Die Klauseln von XQuery lauten FOR, LET, WHERE, ORDER und RETURN, kurz FLWOR (gesprochen: flower) und erinnern stark an die Statements von SQL. FOR und LET entsprechen dem SQL-FOR. In ihnen werden Datenquellen (Dokumente)

selektiert und Stellen in Dokumenten an Variablen gebunden. In der WHERE-Klausel, die dem SQL-WHERE sehr ähnlich ist, können die Variablen zur Spezifikation von Bedingungen verwendet werden. RETURN entspricht dem SQL-SELECT. XQuery bietet in dieser Klausel die Möglichkeit, echtes, verschachteltes XML als Antwort zu generieren.

Während SQL auf Multimengen aufbaut, benutzt XQuery eine Listensemantik. Für jede Anfrage ist die Reihenfolge in der Ausgabe also eindeutig definiert.

Für XQuery existieren auf den Seiten des World Wide Web Konsortiums einige ergänzende Dokumente, die für Anwender und Entwickler sehr empfehlenswert sind. In den XQuery Anwendungsfällen [XQu03b] werden Beispiele für Anfragen in verschiedenen Anwendungsbereichen diskutiert. Hierzu gibt es jeweils konkrete Beispieldokumente, konkrete Anfragen und die als Antwort generierten XML Fragmente.

Eine detaillierte Beschreibung von XQuery würde den Rahmen dieser Arbeit sprengen, weshalb hier auf die einschlägige Literatur insbesondere auf die Internetseiten des W3C verwiesen wird [XQu03a].

Im Bereich von XML Datenbanksystemen kristallisiert sich momentan heraus, dass XQuery zu einem echten und gut angenommenen Standard werden wird. Dies lässt sich schon sagen, obwohl der eigentliche Standard noch nicht verabschiedet wurde. Inzwischen gibt es sogar schon eine Reihe von Implementierungen, wovon ein paar sogar open-source sind [SF04].

2.4.2.3 XOQL

Vor XQuery gab es bereits eine Reihe von XML Anfragesprachen, die im Rahmen verschiedener Projekte entstanden. Die Sprache XOQL [Agu04] wurde aus der Syntax von SQL abgeleitet und stammt demzufolge aus der Datenbankwelt. In der FROM-Klausel werden XML Dokumente referenziert. Auf ihnen können bereits in der FROM-Klausel XPath Ausdrücke angewendet werden, wodurch Knoten in den Dokumenten selektiert werden. Für diese Knoten können dann in der WHERE-Klausel weitere Bedingungen angegeben werden, beispielsweise auch Join-Bedingungen. Weiterhin ist es möglich, strukturierte Ausgabedokumente zu erstellen.

2.5 Konfiguration der Testsysteme

Für Laufzeitmessungen, bei denen der Externspeicherzugriff eine Rolle spielte, wurde ein Rechner verwendet, dessen Festplatte die komplette Kontrolle über die Pufferung ermöglicht. Die Hardware des Rechners sah wie folgt aus:

- AMD Athlon Prozessor 700 MHz mit 1 GB RAM
- Adaptec 2940UW SCSI-Controller
- Festplatte 1: Maxtor 30 GB EIDE
- Festplatte 2: IBM DNES-309170W 9 GB SCSI [IBM99]

Auf Festplatte Nr. 1 wurde das Betriebssystem installiert (Microsoft Windows XP SP1), währenddessen Festplatte Nr. 2 ausschließlich für Tests zur Verfügung stand. Alle untersuchten Verfahren in den folgenden Kapiteln haben einen hohen Anteil an Externspeicherzugriffen. Von daher ist die Architektur der Festplatte von besonderer Bedeutung (siehe Tabelle 2.3).

Bei der IBM DNES 9 GB ist es möglich, den Festplattenpuffer von 1792 KB sowohl für das Lesen, als auch für das Schreiben auszuschalten. Bei den auf Clients gebräuchlicheren EIDE-Festplatten ist dies leider nicht möglich, wodurch alle diese Systeme sich nicht für echte Laufzeittests einsetzen lassen.

Für Messungen, bei denen nur E/A-Operationen gezählt wurden, standen verschiedene Systeme mit Windows und Linux Betriebssystemen zur Verfügung. Deren exakte Konfiguration ist hierbei nicht von Bedeutung.

Bei Messungen der Prozessorlast von Verfahren wurde ein Rechner mit Athlon 2.6 GHz, 1 GB RAM und Windows XP SP1 verwendet. Dieser Rechner war bei Operationen im Hauptspeicher etwa um einen Faktor vier schneller als der Testrechner für die Externspeicherzugriffe.

An Software wurde das JDK 1.4.1 von Sun [Sun02] verwendet. Die XML Parser wurden über die *Java Architecture for XML Processing (JAXP)* angesprochen. Hier wurde der DOM und SAX Parser Apache Xerces [Fou04a] verwendet. Für

Notation	Beschreibung
Reale Kapazität	8,5 GB
Sektorgröße	512
Sektoren pro Spur	247-390
Anzahl Köpfe	5
Anzahl Platten	3
Anzahl Sektoren	17.916.240
Größe des Puffers	1792 KB (Aufteilung in 3 x 512 KB, 7 x 256 KB oder 14 x 128 KB)
Max. Transfergeschwindigkeit	40 MByte/s
Rotationsgeschwindigkeit	7200 RPM
Spurdichte	13700 Spuren pro Inch
Zylinderwechselzeit	2.6 ms (typischer Wert)
Kopfwechselzeit	1.6 ms (typischer Wert)

Tabelle 2.3: Technische Daten der Festplatte IBM DNES-309170W

die Auswertung von XPath-Ausdrücken auf dem DOM-Baum von Xerces wurde Apache Xalan [Fou04b] eingesetzt.

2.6 Allgemeine Notationen

In dieser Arbeit wird eine einheitliche Notation verwendet, die somit teilweise von der Notation der Originalarbeiten abweicht. Deshalb werden an dieser Stelle einige grundlegende Notationen angegeben (Tabelle 2.4).

Notation	Beschreibung
B	Blockgröße auf dem Externspeicher in Bytes
M	Verfügbarer Hauptspeicher in Bytes
$maxRecordSize$	Maximale Größe eines Records. Jedes Record muss in einen Block hineinpassen, so dass $maxRecordSize \leq B$ gilt. Normalerweise ist $maxRecordSize$ nur minimal kleiner als B.
D	Menge von XML Dokumenten
R	XML Baum (Teilbaum oder ganzes Dokument)
$\|R\|$	Anzahl der Elemente in R (Knoten, sowie Attribut- und Textelemente)
$\|\|R\|\|$	Speicherplatz, der von R belegt wird, in Bytes
$k \in R$	Knoten eines XML Baums R
$wurzel(R)$	Wurzelknoten des Baums R
$R.kinder$, $k.kinder$	Folge, die die Kinderknoten vom XML Baum R bzw. des Knotens k enthält.
$kinder[i]$	i-tes Element einer Folge (hier der Kinderknoten). Für das erste Kind sei $i = 1$.
$k.vater$	Vaterknoten von Knoten k (null wenn kein Vaterknoten vorhanden ist)
$T(k)$	Teilbaum beginnend mit dem Knoten k (k ist die Wurzel von $T(k)$)
p	Pfad beginnend bei einem beliebigen, aber festen Knoten
$\|p\|$	Länge des Pfades p.
$h_{max}(R)$	Maximale Höhe des Baums R (Länge des längsten Pfades in R)
$h_{min}(R)$	Minimale Höhe des Baums R. Dies ist die Länge des kürzesten Pfades von der Wurzel von R aus zu einem Blattknoten (der keine Kinder besitzt).

Tabelle 2.4: Notationen

Kapitel 3

Basiskomponenten für Datenbanksysteme

Die Entwicklung von datenintensiven Systemen basiert auf einer Reihe von Kernkomponenten, die die grundlegende Funktionalität für das Gesamtsystem zur Verfügung stellen. Von den Kernkomponenten werden traditionell beispielsweise die folgenden Bereiche abgedeckt: Externspeicherzugriff, Pufferorganisation, Speicherungsstrukturen, Recordmanager, Indexstrukturen und Teile der Anfrageverarbeitung. Die tatsächlich verwendeten Komponenten sind natürlich von System zu System unterschiedlich. In der Datenbankliteratur wurden für jede der genannten Komponenten effiziente Verfahren vorgeschlagen.

Die Implementierung all dieser Komponenten ist sehr aufwendig, weshalb eine Bibliotheksunterstützung unverzichtbar ist. Es wird also eine Bibliothek benötigt, die möglichst viel von der benötigten Funktionalität auf einem hohen fachlichen Niveau anbietet. Zusätzlich sind eine Reihe von Werkzeug-Klassen notwendig, zum Beispiel Hauptspeicherstrukturen wie binäre Bäume, Heaps und vieles mehr. Zugleich sollte eine gute Bibliothek weitere Merkmale aufweisen:

- *Flexibilität*: Es soll einfach möglich sein, Modifikationen durchzuführen.
- *Erweiterbarkeit*: Es soll einfach möglich sein, weitere Verfahren in das Gerüst der Bibliothek zu integrieren.

- *Anwendbarkeit*: Die Möglichkeiten der Bibliothek sollen möglichst breit und einfach anwendbar sein.

- *Effizienz*: Die Verfahren sollen effizient und am Besten worst-case optimal sein.

- *Dokumentation*: Alle Klassen sollen gut dokumentiert sein.

Die wichtigsten Basiskomponenten stehen in der Bibliothek XXL [BBD+01] zur Verfügung, die im Rahmen dieser Arbeit verwendet wurde. Es wäre durchaus möglich gewesen, eine andere Bibliothek wie beispielsweise Berkeley DB [BDB04] zu benutzen. Zum Zeitpunkt der Entscheidung sprach jedoch die vorhandene Funktionalität für XXL. Weiterhin liegen die Stärken von XXL besonders in den Bereichen Erweiterbarkeit und Flexibilität, auf die generell bei der Arbeit großen Wert gelegt wurden.

Als nächstes folgt eine Beschreibung der wichtigsten Pakete von XXL (Kapitel 3.1). Im Rahmen dieser Arbeit wurde XXL um eine ganze Reihe, bislang fehlender Komponenten ergänzt. Diese sind zum großen Teil bereits in der frei verfügbaren XXL Version 1.0 enthalten. Sie werden im Anschluss in Kapitel 3.2 vorgestellt und evaluiert.

3.1 eXtensible and fleXible Library (XXL)

Die Basis dieser Arbeit bietet die plattformunabhängige Java-Bibliothek XXL (*eXtensible and fleXible Library*), die von der Arbeitsgruppe Datenbanksysteme der Philipps-Universität Marburg unter Leitung von Professor Dr. Bernhard Seeger entwickelt wurde. Die Bibliothek stellt eine reiche Infrastruktur für die Entwicklung von Datenbanksystemen und Datenbankapplikationen zur Verfügung. Weiterhin stehen Beispielapplikationen zur Verfügung. Alle Klassen integrieren sich nahtlos in das SDK von Java. XXL ist frei auf der Internetseite des Projekts erhältlich. Die Bibliothek steht unter der LGPL-Lizenz [GNU], d. h. sie darf auch in kommerziellen Projekten frei verwendet werden.

Grundlegend ist in XXL die Wahrung des Prinzips der physischen Datenunabhängigkeit, so dass die Anfragefunktionalität unabhängig von der tatsächlichen Speicherung bleibt. Hierdurch eignet sich die Bibliothek auch als Middleware für Datenbanksysteme. Im Gegensatz zu klassischen, monolithisch aufgebauten Datenbanksystemen kann durch den Bibliotheksansatz eigene Funktionalität einfach integriert werden. XXL kann dann als eine Art eingebettetes Datenbanksystem in Applikationen benutzt werden.

Zum Zeitpunkt dieser Arbeit liegt die Bibliothek in der Version 1.1b1 vor. Aktuelle Versionen, eine vollständige Dokumentation der API (*Application Programming Interface*) und weitere Informationen finden sich auf der Internetseite des Projekts [XXL].

Die Bibliotheksklassen sind in verschiedene zweckgebundene Java-Pakete unterhalb von `xxl.core` aufgeteilt. Die zentralen Pakete sind `cursors`, `collections`, `io`, `functions` und `predicates`. Die Zuordnung von Klassen zu den Paketen ist nicht ganz eindeutig, da es beispielsweise Collection-Klassen gibt, die `io`-Aspekte beinhalten. In diesem angesprochenen Fall wurde die Entscheidung getroffen, sie unterhalb von `collections` in ein Paket `io` einzuordnen.

Die genannten fünf Pakete (`predicates`, `functions`, `cursors`, `collections` und `io`) stellen die Basis von XXL dar. Die dort definierten Schnittstellen und Klassen finden überall in der gesamten Bibliothek breite Verwendung. Die Konzepte der Basis sind so allgemein gehalten, dass sich die Klassen für viele verschiedene Zwecke einsetzen lassen.

Im Folgenden werden die wichtigsten Prinzipien der Bibliothek erläutert, die zum Verständnis der im Rahmen dieser Arbeit implementierten Verfahren notwendig sind.

3.1.1 Funktionen und Prädikate

Grundlegende Konzepte von XXL sind Funktionen (Klasse `Function`) und Prädikate (Klasse `Predicate`). Funktionen sind Java-Objekte, die von `xxl.core.functions.Function` erben. Unter einer Funktion ist im Wesentlichen das zu

verstehen, was auch ein Mathematiker unter einer Funktion versteht, also eine Abbildung von einer Menge A in eine Menge B. Allerdings können Funktionen in XXL auch von einem Zustand abhängen, wovon gelegentlich Gebrauch gemacht wird. In XXL werden Funktionsaufrufe mit beliebiger Stelligkeit unterstützt. In einer von `Function` erbenden Klasse kann entweder die Methode für den allgemeinen n-stelligen Funktionsaufruf überschrieben werden, oder es muss mindestens einer der speziell vordefinierten null-, ein- oder zweistelligen Funktionsaufrufe überschrieben werden (siehe Abbildung 3.1). Mit Hilfe eines in der Klasse `Function` integrierten Rekursionsmechanismus werden Aufrufe von nicht überschriebenen Methoden an die implementierten Methoden gleicher Stelligkeit weitergereicht.

```
public Object invoke();
public Object invoke(Object argument);
public Object invoke(Object arguments, Object argument2);
public Object invoke(Object[] arguments);
```

Abbildung 3.1: Die `invoke`-Methoden der Klasse `Function`

Durch Komposition von Funktionen (Methode `compose` in Function) können Funktionen höherer Ordnung definiert werden. Dies soll hier an einem kurzen Beispiel erläutert werden. Gegeben sind die Funktionsklassen `Sinus`, `Cosinus` und `Div`. Es existieren weiterhin konkrete Instanzen dieser Klassen namens `sin`, `cos` und `div`. Dann kann der Tangens (tan) einfach durch Komposition implementiert werden (siehe Abbildung 3.2).

```
Function tan = div.compose(sin, cos);
```

Abbildung 3.2: Beispiel für die Komposition von Funktionen

Funktionen werden sehr breit in allen Paketen von XXL verwendet. Viele Klassen werden mit Funktionen parametrisiert, z. B. mit Objektfabriken [GHJV95]. Gelegentlich erinnert die Verwendung von Funktionen an call-back Funktionen in C++. Durch die allgemein gehaltene Definition von Funktionen, können diese nahezu universell eingesetzt werden. Wenn gewisse Klassen von Implementierungsdetails abstrahieren wollen, dann verwenden sie häufig Funktionen. Dadurch können einmal implementierte Klassen in neuen, vorher eventuell nicht

bedachten Kontexten eingesetzt werden. Dies ist ein wesentlicher Grund für die hohe Flexibilität von XXL.

In vielen Situationen ist es sinnvoll, Funktionen mittels anonymer Klassen zu definieren. Dies hat den Vorteil, dass eine Funktion an der Stelle definiert wird, an der sie gebraucht wird. Dadurch kann die Zahl der Klassen reduziert und die Übersichtlichkeit des Quelltextes erhöht werden.

Aufgrund der zentralen Bedeutung von Funktionen existieren eine Reihe von allgemein verwendbaren, vordefinierten Funktionen im Paket `xxl.core.functions`. Diese Funktionen orientieren sich an bekannten Prinzipien aus dem Bereich der funktionalen Programmierung.

Prädikate sind eine Sonderform des allgemeineren Prinzips der Funktionen. Prädikate unterscheiden sich nur dadurch, dass sie ausschließlich boolesche Werte zurückgeben. Prädikate bewirken mehr Typsicherheit in Anwendungen, da der Rückgabewert von Funktionen immer erst in den erwarteten Datentyp umgewandelt werden muss. Da die Prüfung der Korrektheit dieser Umwandlung i. Allg. erst zur Laufzeit erfolgen kann, sind Laufzeitfehler möglich und durchaus problematisch. Der Einsatz von Prädikaten bewirkt weiterhin einen Gewinn an semantischer Klarheit und nicht zuletzt an Geschwindigkeit, da keine echten Java-Objekte zur Rückgabe erzeugt werden müssen.

Prädikate können mittels einer so genannten *Wrapper-Klasse* in Funktionen mit booleschem Rückgabetyp umgewandelt werden. Analog kann die Umwandlung von booleschen Funktionen in Prädikate erfolgen. Hierdurch können Standard-Funktionen auf Prädikate angewendet werden. Weiterhin gibt es spezielle Prädikatsklassen wie `And`, `Or` und `Not`, die für allgemeine Funktionen nicht sinnvoll sind.

3.1.2 Datenströme

Die effiziente Verarbeitung von Datenströmen nimmt in Datenbanksystemen einen großen Stellenwert ein. Eine Anfrage in einer Datenbankanfragesprache wie SQL löst beispielsweise einen oder mehrere Datenströme aus. Neben dem

eigentlichen Datenstrom spielen Metainformationen über den Typ der Daten im Strom eine wichtige Rolle. Metadaten sind allerdings, so weit dies geht, von der Verarbeitung der Elemente des Datenstroms zu trennen, weil dadurch eine größere Flexibilität erreicht wird. Zusätzlich muss von den zugrunde liegenden Datenquellen (Relationen, Dateien, URLs, etc.) abstrahiert werden.

Java stellt mit dem Konzept der *Iteratoren* eine Abstraktion zur Verarbeitung von anfragegesteuerten, passiven[1] Datenströmen (*demand-driven data stream*) zur Verfügung. Die Schnittstelle des Iterators ist in Abbildung 3.3 zu finden.

```
interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}
```

Abbildung 3.3: Die Schnittstelle Iterator von Java

Traditionell werden in Datenbanksystemen ONC-Operatoren (open-next-close [Gra93]) eingesetzt. Wenn man dieses Modell auf Java-Iteratoren überträgt, so muss die *open*-Phase implizit nach der Instanzierung eines Iterator-Objektes ausgeführt werden. Die *next*-Phase wird durch die gleichnamige Methode in `Iterator` realisiert. Dabei zeigt die Methode `hasNext` an, ob weitere Objekte im Datenstrom verfügbar sind. Die *close*-Phase ist sehr viel problematischer, weil zum einen die Schnittstelle `Iterator` kein close anbietet und Java zum anderen keine Destruktoren besitzt, die garantiert nach dem Ableben eines Objekts gerufen werden[2]. Somit können Iteratoren in dieser Form nicht in Datenbanksystemen Verwendung finden, da ein explizites close bei Dateien und anderen Quellen unverzichtbar ist.

Dafür bietet `Iterator` die optionale Methode `remove()` an, mit der das letzte mit `next()` gelieferte Objekt in der zugrunde liegenden Datenquelle gelöscht wird. Dies stellt quasi eine Erweiterung des ONC-Prinzips dar.

[1] Passiv bedeutet hier, dass der Datenstrom von sich aus keine Objekte an einen Verbraucher sendet.

[2] Es gibt keine Garantien dafür, dass eine überschriebene `finalize`-Methode irgendwann aufgerufen wird. Wenn ein Aufruf erfolgt, so kann dieser zeitlich nicht terminiert werden.

In XXL findet die Schnittstelle `Cursor` des Pakets `xxl.core.cursors` Verwendung, die von `Iterator` erbt und u. a. eine explizite `open`- und `close`-Phase unterstützt (siehe Abbildung 3.4).

```
interface Cursor extends Iterator {
    void open();
    void close();
    Object peek();
    void update(Object o);
    void reset();

    boolean supportsPeek();
    boolean supportsReset();
    boolean supportsUpdate();
    boolean supportsRemove();
}
```

Abbildung 3.4: Die Cursor-Schnittstelle in XXL

Hinzugekommen ist in Cursor die `peek`-Methode, die das nächste Element liefert, ohne es allerdings endgültig zu entnehmen. Erst der nächste Aufruf von `next` entnimmt das Element. Bei einigen Algorithmen bringt die `peek`-Methode Vorteile in der Implementierung.

Die Methode `reset` setzt den Cursor wieder an den Anfang der Daten, die geliefert wurden. Es kann dann ein weiteres Mal über die Daten gegangen werden. Mit `update` kann das letzte Element in der zugrunde liegenden Datenquelle verändert werden.

In Cursor sind die Methoden `peek`, `update`, `reset` und `remove` optional. Wenn ein Cursor sie nicht implementieren kann oder will, so wirft er eine Ausnahme (*Exception*). Damit sich eine Applikation auf die Fähigkeiten eines Cursors einstellen kann, gibt es die Methoden mit Präfix `supports`. Hier kann die Applikation fragen, ob die Methoden generell implementiert sind oder nicht. Es kann natürlich trotzdem sein, dass ein `update` fehlschlägt, weil das Element in der Datenquelle nicht eindeutig identifiziert werden kann.

Das Paket `xxl.core.cursors` stellt eine Vielzahl von *Cursors* für verschiedene Datenquellen, sowie zur Erzeugung und Verarbeitung von Daten zur Verfügung. Es ist eine vollständige Algebra zur Anfrageverarbeitung mit benutzerdefinierten Funktionen und Prädikaten enthalten. Wichtige Cursor sind:

- *Filter*: Filtert Elemente aus einem Datenstrom unter Anwendung eines Prädikats heraus (entspricht einer WHERE-Klausel in SQL).
- *Mapper*: Transformiert die Elemente eines Datenstroms mittels einer gegebenen Funktion.
- *Grouper*: Gruppiert die Elemente eines Datenstroms nach gewissen Parametern und stellt für jede Gruppe wieder einen Datenstrom zur Verfügung (entspricht einem *nest*). Es gibt mehrere Grouper, die mit verschiedenen Prinzipien arbeiten (nested loops, sort-based, hashing).
- *Sequentializer*: Fügt mehrere Datenströme mit Elementen zusammen. Dies können beispielsweise die Gruppen von einem Grouper sein (entspricht einem *unnest*).
- *Aggregator*: Berechnet für eine Gruppe einen Aggregatwert.
- *Sorter*: Sortiert die Datenmenge nach einem vorgegebenen *Comparator*.

Das Paket `xxl.core.cursors` stellt die Grundfunktionalität für die Anfrageverarbeitung in XXL dar. In den folgenden Abschnitten soll auf einige Anwendungen eingegangen werden. `xxl.core.cursors` enthält nur Algorithmen, die als Eingabe Daten vom Typ Java-`Object` erhalten und eben solche als Ausgabe produzieren. Die Cursor sind also so weit es geht allgemein gehalten und besitzen keinerlei Metadaten. Aufbauend auf dieses Grundkonzept gibt es das Paket `xxl.core.relational`, das ausschließlich relationale Tupel verarbeitet. Jeder Cursor ist hier ein `MetaDataCursor`, der relationale Metadaten basierend auf der JDBC 2.0 Spezifikation besitzt. Im Paket `xxl.core.xml.operators` existiert eine weitere Art von Operatoren, die auf XML Basis arbeitet und entsprechend völlig andere Metadaten besitzt. Auf diese Operatoren wird in Kapitel 4 noch ausführlich eingegangen.

In Datenbanksystemen wird häufig unterschieden zwischen logischen und physischen Operatoren [Gra93]. In diesem Sinn sind alle bisher angesprochenen Operatoren physisch. Für Optimierungszwecke sind jedoch logische Operatoren sehr interessant. Diese sind im Paket `xxl.core.relational.optimization` vorhanden. Bei logischen, relationalen Operatorbäumen lassen sich Optimierungen mittels Heuristiken vornehmen, die auf einfachen Regeln der relationalen Algebra basieren. Nach diesem ersten Optimierungsschritt werden die logischen Bäume unter Zuhilfenahme von Selektivitätsschätzungen und weiteren Techniken in physische Operatorbäume übersetzt.

Alle in XXL integrierten Implementierungen sind soweit möglich *bedarfsgesteuert*, d. h. ein Zugriff auf die Datenquelle bzw. eine Manipulation der Daten erfolgt erst bei der expliziten Anforderung der Daten durch die Applikation. Völlig komplementär dazu ist der Ansatz, dass Datenquellen aktiv sind und Daten selbstständig an Operatoren senden. Eine solche Herangehensweise ist im Paket `xxl.core.pipes` realisiert [KS04]. Operatoren müssen sich in PIPES bei den Datenquellen anmelden und erhalten die Daten dann, sobald sie vorliegen. Hierdurch werden dauerhafte Anfragen (Continuous queries, CQ) ermöglicht, die jahrelang auf einem Server aktiv sein können, immer wieder Eingabedaten erhalten und Ergebnisse liefern. Entsprechende Verfahren finden bei den so genannten Datenstrom Management Systemen (DSMS) Anwendung.

3.1.3 Zugriff auf den Externspeicher

Das Paket `xxl.core.io` enthält die Funktionalität, die für den Zugriff auf den Externspeicher wesentlich ist. In Java gibt es keine Möglichkeit, die Speicherung von Objekten auf das Byte genau zu definieren. Die Java-Serialisierung legt beim ersten Speichern des Objekts einer gewissen Klasse immer noch Typinformationen mit ab. Jeder Block auf dem Externspeicher würde somit die gleiche Typinformation enthalten, was unter dem Gesichtspunkt der Speicherplatzausnutzung nicht vertretbar wäre.

Für die Lösung dieses Problems gibt es in XXL die so genannten *Converter*. Diese werden eingesetzt, um Objekte in eine Externspeicherdarstellung

zu überführen und aus dieser Darstellung umgekehrt das gleiche Objekt wieder herzustellen. Im Unterpaket `xxl.core.io.converters` gibt es bereits eine Reihe von sofort nutzbaren Convertern für die wichtigsten Datentypen. Ein weiterer Vorteil von Convertern im Vergleich zur Java Serialisierung ist, dass man mehrere Converter für Objekte einer Klasse definieren kann. Je nach Verwendungszweck können unterschiedliche, speziell optimierte Converter eingesetzt werden, was den Datenverkehr minimiert.

Ein weiterer, wichtiger Teil des `io`-Pakets ist das Puffermanagement. Wie bereits in Kapitel 2.3 ausgeführt, sind die Zugriffszeiten von heutigen Festplatten ein großes Problem, da sie die Prozessoren ausbremsen. Um das Problem zu entschärfen, werden auf verschiedenen Ebenen Puffer eingesetzt, welche die Daten, auf die zuletzt zugegriffen wurde, im Hauptspeicher halten. Normaler Weise werden hierfür Puffer mit LRU-Strategie (Least Recently Used) eingesetzt. In XXL ist eine allgemein einsetzbare Pufferklasse enthalten, die mit einer gewünschten Strategie parametrisiert werden kann.

3.1.4 Collections

Viele Algorithmen in Datenbanksystemen benötigen eine Infrastruktur bestehend aus Standarddatentypen wie Warteschlangen (Queues), Heaps und Containern. Das Paket `xxl.core.collections` stellt solche grundlegenden Datenstrukturen zur Verfügung.

Auf die Betrachtung von Warteschlangen und Heaps soll an dieser Stelle verzichtet werden. Jedoch ist das Containerkonzept von großer Bedeutung für diese Arbeit und verdient daher genauere Erläuterung. Ein Container speichert ganz allgemein betrachtet Objekte, die ihm übergeben werden. Um die Objekte wieder auffinden zu können, erhält man beim Einfügen einen Identifikator, mit dem das Objekt wieder gefunden werden kann. Das Prinzip ist dasselbe wie bei einer Garderobe im Theater, wo man für seinen Mantel einen Zettel mit einer Nummer erhält. Nach der Aufführung wird der Mantel gegen Vorlage des Zettels zurück gegeben.

Ein Container verwaltet daher die Abbildung (`Map`) von Identifikatoren zu den Objekten. Die Schnittstelle `Container` erlaubt das Einfügen von Objekten, das Herausgeben von Objekten mittels Identifikatoren, sowie das Verändern und Löschen der gespeicherten Objekte. Es können weiterhin alle verwalteten Objekte, sowie alle Identifikatoren per Iterator zurückgegeben werden.

In XXL gibt es verschiedene Container-Implementierungen. Die einfachste Implementierung wird realisiert mit einer Java-Map (`java.util.Map`). Hierbei werden die Objekte im Hauptspeicher gehalten. Für Datenbanksysteme sind selbstverständlich externspeicherfähige Container wichtig. Hier gibt es den `BlockFileContainer`, der auf Dateien operiert und Blöcke mit einer vorgegebenen maximalen Größe verwaltet. Diese Klasse enthält als zentralen Bestandteil eine Freispeicherverwaltung, so dass freigegebene Blöcke wieder genutzt werden können.

Da ein `BlockFileContainer` nur mit Blöcken umgehen kann und nicht mit allgemeinen Objekten, ist eine Umsetzung nötig. Hierfür gibt es den `ConverterContainer`, der auf der Basis eines beliebigen Containers operiert (Prinzip der Dekoration [GHJV95]) und die Konvertierung mit einem gegebenen Converter durchführt.

Für die Pufferung der Objekte eines Containers existiert die Klasse `BufferedContainer`, die ebenso einen vorhandenen Container dekoriert. Zunächst wird immer versucht, die Anfrage aus einem gegebenen Puffer zu beantworten. Erst danach wird der zugrunde liegende Container befragt. Ein BufferedContainer kann sowohl unterhalb eines ConverterContainers, als auch oberhalb eingesetzt werden. Im ersten Fall handelt es sich um einen Blockpuffer, im zweiten Fall um einen Objektpuffer.

3.1.5 Zusätzliche Funktionalität in XXL

Im Paket `xxl.core.indexStructures` befinden sich Implementierungen für verschiedene externspeichertaugliche Indexstrukturen. Unter anderem sind dort ein B+-Baum, mehrere R-Baum-Varianten, ein X-Baum und zwei M-Baum Varianten zu finden. Das zugrundeliegende Framework in den Klassen Tree

und ORTree ist generisch gehalten, so dass es hilfreich für die Implementierung vieler verschiedener Indexstrukturen ist.

Die Pakete `xxl.core.spatial`, `xxl.core.binarySearchTrees` und `xxl.core.math` enthalten Klassen und Schnittstellen, die ebenfalls gebräuchlich im Datenbankumfeld sind (Funktionalität für Geo-Daten, binäre Hauptspeicherbäume, mathematische Funktionen). In `xxl.core.util` finden sich weitere Hilfsmittel, die keinem der anderen Pakete inhaltlich zuzuordnen sind.

3.2 Erweiterung der Funktionalität von XXL

In den folgenden Unterkapiteln geht es um allgemein nutzbare Funktionalität, die für die Fertigstellung dieser Arbeit benötigt wurde, bisher allerdings in XXL noch nicht verfügbar war. Alle im folgenden beschriebenen Funktionalitäten wurden im Rahmen dieser Arbeit implementiert, dokumentiert und nahtlos in XXL integriert. Es wurde entsprechend der Philosophie von XXL Wert auf einfache und weitreichende Parametrisierbarkeit gelegt, kurz gesagt auf Generizität.

Soweit die im folgenden beschriebenen Komponenten noch nicht in der aktuellen auf der Internetseite bereitgestellten XXL Version vorhanden sind, wird deren Veröffentlichung mit der nächsten Version erfolgen.

3.2.1 Das Test-Framework

XXL empfiehlt sich als *Testplattform*, da in ihr bereits viele Algorithmen enthalten sind, die allgemein verwendbar sind. Die Implementierung neuer Verfahren kann somit in kürzerer Zeit erfolgen als dies in der Vergangenheit möglich war.

Um das Testen von neuen Algorithmen zu erleichtern, wird im Test-Framework die Möglichkeit geboten, Testfälle komfortabel zu konstruieren. Dazu müssen die Parameter des Testfalls nur als Klassenvariablen der Testklasse nach außen geführt werden. Für jeden Parameter können minimal und maximal mögliche

Werte in weiteren Klassenvariablen angegeben werden. Diese Klassenvariablen müssen `final` sein, den selben Datentyp wie der Parameter besitzen und der Name muss mit Namen des Parameters beginnen und mit „Min“, bzw. „Max“ enden. Hieraus können bei `integer`- und `double`-Typen automatisch Testwerte generiert werden. Die Testwerte können auch direkt angegeben werden, indem in der Klasse ein Feld mit dem Namen des Parameters plus „Values“ angelegt wird, welches eine Menge von Testwerten enthält.

Der *Testfallgenerator* erzeugt alle Kombinationen von Werten, die für alle Parameter eines Testfalls möglich sind (also das Kreuzprodukt). Der eigentliche *Tester* ist ein `MetaDataCursor`, der die erzeugten Testwerte in Form eines `MetaDataCursor`s bekommt. Er belegt die Testklasse mit den gewünschten Werten, ruft die `main`-Methode auf und gibt die Testergebnisse bedarfsgesteuert zurück. Zusätzlich zu den normalen Testergebnissen (Zeitmessungen, Zählung von Zugriffen, etc.) werden standardmäßig eine Reihe von wichtigen Daten mit zurück gegeben, wie beispielsweise Datum und Zeit des Tests, der Rechnername oder eventuell aufgetretene Ausnahmesituationen. Mit Hilfe von JDBC können die Testergebnisse direkt in eine Datenbank geschrieben werden, wo sie zur Auswertung zur Verfügung stehen.

3.2.2 Festplattenzugriff

Algorithmen, die für den Externspeicher entworfen wurden, müssen in theoretischen Betrachtungen und in Laufzeittests ihre Tauglichkeit beweisen. Bei den Laufzeittests können jedoch Komponenten des Betriebssystems eine entscheidende Rolle spielen. Um den Unterschied in den Zugriffsgeschwindigkeiten zwischen Haupt- und Externspeicher zu mildern, verwenden Betriebssysteme den freien Hauptspeicher, um Daten zu puffern. Bei heutigen Hauptspeichergrößen von 1GB und mehr kann der Seitenpuffer schon einige hundert Megabyte groß sein.

Wenn beim Test eines neuen Externspeicheralgorithmuses nur wenig Hauptspeicher direkt allokiert wird, so kann das Betriebssystem den Rest des Hauptspeichers für den Puffer verwenden. Die gemessene Leistung hängt dann nicht

mehr vom Externspeicher, sondern wesentlich von der Pufferleistung ab. Wenn der zu testende Algorithmus in der Praxis dann aber parallel zu 100 anderen Algorithmen auf einem Server arbeitet, so passen seine Daten wahrscheinlich nicht mehr in den Puffer und die Leistung bricht unvorhersagbar ein. Der Test eines Algorithmus muss also in einer Umgebung erfolgen, die dem späteren Einsatzszenario möglichst nahe kommt.

Wird der Algorithmus später auf einem Server arbeiten, was in unseren Szenarien auch so ist, so müssen die Puffergrößen beim Test entsprechend realistisch gewählt werden. Diese Wahl ist in modernen Betriebssystem (Unix-Derivate oder Windows) leider nicht möglich.

Ein weiteres Problem ist, dass der Betriebssystempuffer die Daten nicht sofort auf die Festplatte schreibt, sondern das Wegschreiben erst nach einiger Zeit asynchron durchführt. Eine Applikation weiß somit nicht, wann die Daten wirklich persistent auf den Magnetscheiben der Platte vorliegen. Dies ist für Datenbanksysteme nicht akzeptabel, da bei *Write Ahead Logging* bekannt sein muss, wann die Daten definitiv persistent sind.

Probleme gibt es weiterhin mit dem Puffer, der in modernen Festplatten eingebaut ist. Der Festplattenpuffer ist deutlich kleiner als der Seitenpuffer des Betriebssystems (typischer Weise 8 MB) und wird i. d. R. auf eine andere Art und Weise verwaltet. Priorität hat hier, dass bei einem Zugriff gleich die nächsten Sektoren mit gelesen werden (*Read-Ahead*), da ein Zugriff auf diese Sektoren in der Folgezeit sehr wahrscheinlich ist (sequential read). Ohne den Festplattenpuffer müsste bereits beim leicht zeitversetzten Lesen des nächsten Sektors nahezu eine volle Umdrehung gewartet werden, was nicht akzeptabel ist. Da der Puffer auf der Festplatte selbst verwaltet werden muss, steht für den Pufferalgorithmus keine so leistungsfähige Hardware zur Verfügung wie im Rechner selbst. Aus diesen Gründen werden im Puffer immer nur sehr große Blöcke gehalten. Bei der im Test verwendeten Festplatte [IBM99] mit 1792 KB Puffer wird der Puffer in drei bis 14 Segmente zerlegt, die immer nur zusammenhängend als Schreibpuffer oder für Read-Ahead benutzt werden.

Nicht vergessen werden darf, dass es auch spezielle Puffer in Funktionen von Bibliotheken geben kann. Bei der Standardbibliothek von C (`fopen`, `fseek`,

`fread`, `fwrite`, `fclose`) lässt sich der Puffer einfach mit dem Kommando `setbuf` ausschalten. Bevor Bibliotheken in Algorithmen verwendet werden, sind also deren Pufferungsmöglichkeiten zu prüfen.

Bei der Lösung der genannten Probleme mit den unkontrollierbaren Puffern erweist es sich als kontraproduktiv, in einer höheren Programmiersprache zu arbeiten. Je abstrakter die Programmiersprache ist, desto weniger Kontrolle hat der Programmierer über die einzelnen hardwaremäßig vorhandenen Komponenten. Diese Kontrolle muss dann über Umwege zurück erlangt werden.

Die Programmiersprache Java läuft auf einer virtuellen Maschine, die eine eingeschränkte Funktionalität des darunter liegenden Rechners zur Verfügung stellt. Die Java-VM besitzt keine Möglichkeit, direkt auf die Pufferung des Betriebssystems Einfluss zu nehmen. Das WAL-Problem ist ebenfalls nicht in Java selbst zufriedenstellend lösbar.

Wegen solcher Probleme könnte man sagen, dass ein Datenbanksystem besser in einer hardwarenahen Sprache zu implementieren sei. Die Folge hiervon wäre jedoch, dass die komplexen Funktionen deutlich längere Entwicklungszeiten benötigten.

In XXL hat sich gezeigt, dass es wenige Komponenten gibt, die hardwarenah implementiert werden müssen. An den Stellen, wo dies nötig ist, wurde C-Code in plattformspezifischen Bibliotheken (DLL für Windows und `.so`-Datei für Linux) über JNI (Java Native Interface) an XXL angebunden. Bislang gibt es zwei nativ implementierte Bereiche:

- eine deutlich verbesserte Zeitmessung (die für Messungen im oberen Mikrosekundenbereich nötig ist, auf die hier allerdings nicht weiter eingegangen werden soll).
- den direkten Zugriff auf den Externspeicher (Raw-E/A).

3.2.2.1 Direkter Festplattenzugriff

Der direkte Externspeicherzugriff erlaubt die volle Kontrolle über die Festplatte. Alle Lese- und Schreiboperationen werden ohne Umweg über den Be-

triebssystempuffer direkt an die Festplatte weitergeleitet. Weiterhin ist es unter Windows bei einigen Festplatten möglich, den Festplattenpuffer auszuschalten. Bei der verwendeten IBM-Festplatte [IBM99] ist es möglich, sowohl den Lese-, als auch den Schreibpuffer der Platte getrennt voneinander auszuschalten. Bei vielen IDE-Festplatten ist ein Ausschalten des Lesepuffers leider nicht möglich.

Was gepuffert wird, liegt bei geeigneter Festplatte somit vollkommen in der Hand des Entwicklers. Zu beachten ist jedoch: Wird der Lesepuffer der Festplatte ausgeschaltet, so wird jeder sequentielle Festplattenzugriff zu einem extrem langsamen Zugriff, bei dem die Platte fast eine komplette Umdrehung zurücklegen muss ($t_{seq} \approx 60/RPM$).

Die Verbindung des direkten Festplattenzugriffs zu XXL wurde über eine einfache Schnittstelle namens `RawAccess` realisiert (Abbildung 3.5). Diese erlaubt einen Blockzugriff auf den Externspeicher.

```
public interface RawAccess {
    public void open(String filename);
    public void close();
    public void write(byte[] block, long sector);
    public void read(byte[] block, long sector);
    public long getNumSectors();
    public int getSectorSize();
}
```

Abbildung 3.5: Die RawAccess-Schnittstelle in XXL

Eine Festplatte wird also in eine Reihe von Sektoren gleicher Größe eingeteilt (normaler Weise 512 Bytes, aber variabel in der Schnittstelle). Anstatt der dreidimensionalen Adressierung mit Zylinder, Kopf und Sektor (siehe Kapitel 2.3), wird hier eine eindimensionale Adressierung verwendet. Die Sektoren sind von 0 bis `getNumSectors()`-1 durchnummeriert, wobei sie lexikographisch nach Zylinder- (von außen nach innen), Kopf- und Sektornummer sortiert sind.

Wenn die Platte auf Sektor n steht und anschließend Sektor $n + 1$ gelesen werden soll, so fällt i. d. R. nur die Rotationszeit über den zu lesenden Sektor als Verzögerungszeit an. In einigen Fällen wird zusätzlich eine Kopfwechsel-

zeit (*Head-Switch-Time*) benötigt, in weiteren relativ seltenen Fällen muss der Schreib-/Lesekopf auf den nächsten Zylinder positioniert werden. Allerdings gibt es auch den umgekehrten Fall, dass keine Verzögerung auftritt, obwohl sich die Sektornummern um mehr als eins voneinander unterscheiden. Dies liegt daran, dass die Sektoren in Kreisen angeordnet sind. Wenn auf einer Spur s Sektoren vorhanden sind, so kann nach Sektor $s-1$ der Sektor 0 dieser Spur ohne Kopfwechselzeit und Neupositionierung gelesen werden. Für diese Arbeit ist es jedoch ausreichend davon auszugehen, dass ein Zugriff sequentiell erfolgt, wenn die Sektornummer vom aktuellen Zugriff genau um eins höher ist, als die des vorherigen Zugriffs. Ansonsten wird der Zugriff als wahlfrei angesehen.

Sektoren werden in Bytefelder der Größe des Sektors eingelesen, und solche Bytefelder können wieder als Sektoren auf die Platte geschrieben werden. Dies ist sehr effizient.

Die Schnittstelle `RawAccess` kann auf Dateien, Partitionen oder ganze Festplatten angewendet werden. Um die Anzahl der sequentiellen, bzw. wahlfreien Zugriffe zu ermitteln, kann das benutzte `RawAccess` bei Bedarf in ein statistisches Dekoratorobjekt eingepackt werden. Weiterhin existiert eine Implementierung der `RawAccess` Schnittstelle, die alle Zugriffe auf einen Hauptspeicherbereich umleitet. Somit kann in Testläufen die Anzahl der Seeks ermittelt werden, ohne den Algorithmus auf einer Festplatte laufen lassen zu müssen. Dadurch werden Tests enorm beschleunigt. Die Klasse `NativeRawAccess` implementiert die `RawAccess` Schnittstelle nativ in der Programmiersprache C.

Bei der Windows Implementierung werden low-level Funktionen der Windows API benutzt. Beim Kommando `CreateFile` können die entscheidenden Optionen zur Pufferung gesetzt werden. Über die Dateinamen „\\.\X:“ können Partitionen oder ganze Festplatten angesprochen werden. `NativeRawAccess` besitzt zusätzlich eine Methode mit dem Namen `setHardDriveCacheMode`. Mit ihr ist es möglich, den Puffer der Festplatte getrennt für Lese- und Schreiboperationen ein- und auszuschalten.

Unter Linux werden vorhandene Partitionen oder Festplatten benutzt, die nicht in den Verzeichnisbaum eingebunden wurden und somit roh zur Verfügung stehen. Für den Zugriff können dann die Routinen aus der Standardbibliothek

verwendet werden. Eine einfache und von der Hardware unabhängige Möglichkeit zur Abschaltung der Puffer wurde in Linux bislang nicht gefunden.

Somit erhält die Windows-Implementierung bislang den Vorzug, wenn ein Einfluss des Festplattenpuffers ausgeschlossen werden soll.

3.2.2.2 Evaluierung

Die implementierten Zugriffsmethoden mussten beweisen, dass sie wie erwartet funktionieren. Dazu wurde eine Reihe von Experimenten durchgeführt. Zunächst geht es um die Überprüfung, ob die Pufferung wirklich ausgeschaltet werden konnte. Anschließend wird die Leistung von sequentiellen Zugriffen getestet und mit theoretisch ermittelten Werten verglichen.

Puffer

Zunächst wurde überprüft, ob tatsächlich alle Puffer umgangen werden konnten. Dazu wurde der erste Sektor der Festplatte mehrfach hintereinander gelesen. Jede Pufferstrategie, sowohl die des Betriebssystems, als auch die der Festplatte, würde den Sektor beim ersten Lesen puffern und anschließend alle Anfragen aus dem Puffer in einem Bruchteil der Zeit beantworten.

Das 1000-fache Lesen des ersten Sektors benötigte $8,3s$. Dies ist ziemlich genau die Zeit, die für 1000 Umdrehungen der Festplatte benötigt werden. Es zeigte sich weiterhin, dass das sequentielle Lesen von zwei Sektoren etwas länger als eine Rotation der Festplatte dauerte. Dies war erwartet worden, weil sich zwischen dem Ende des Transfer des ersten Sektors und der Anforderung, den nächsten Sektors zu lesen, die Oberflächen weiter drehen. Somit stehen die Leseköpfe zum Zeitpunkt der Anforderung bereits jenseits des Anfangs des nächsten Sektors, wenn auch nur minimal. Da der Festplattenpuffer ausgeschaltet ist und daher kein *Read-ahead* erfolgen kann, muss mit dem Lesen des zweiten Sektors nahezu eine Umdrehung gewartet werden.

Setzt man zum sequentiellen Lesen mehrere Threads ein, so ist es grundsätzlich möglich die zusätzliche Rotation einzusparen. In Fällen, in denen der erste

gelesene Sektor ausgewertet werden muss, bevor der nächste Sektor angefordert werden kann, helfen auch mehrere Threads und sogar mehrere Prozessoren nicht weiter. Hier muss selbst bei Zugriffen auf den nächsten Sektor eine Rotation lang gewartet werden.

Das Fazit dieser Experimente ist, dass die `RawAccess`-Schnittstelle und ihre Implementierung in C unter Windows in der Tat alle Puffer von Betriebssystem und Festplatte umgehen kann und in so fern eine gute Basis für Experimente darstellt.

Sequentielles Lesen

Eine gewisse Unsicherheit stellt die Benutzung des *Java Native Interfaces* dar. Die Operationen im C-Quelltext müssen aufwendige Typkonvertierungen vornehmen. Außerdem kosten Zugriffe auf Attribute der Objekte mehr Zeit als in C, da die Werte über Methodenaufrufe gelesen, bzw. geschrieben werden müssen.

Das zweite Experiment soll daher zeigen, dass die maximal erzielbaren Datenraten über die `RawAccess`-Schnittstelle auf einem angemessenen Niveau und nicht weit weg von einer reinen C Implementierung liegen. Um die Leistungsfähigkeit von JNI zu prüfen, wurde die Festplatte im Modus mit maximalem Durchsatz betrieben, d. h. bei sequentiellem Lesen und eingeschaltetem *Read-ahead* Festplattenpuffer. Der Testaufbau sah so aus, dass n Blöcke der Größe s Sektoren sequentiell von Sektor 0 der Festplatte ausgehend gelesen wurden. Aus der gemessenen Zeit lässt sich die Datentransferrate in MB/s ermitteln. Der Parameter n wurde auf einen hinreichend großen Wert festgesetzt ($n = 1000$), und der Parameter s wurde variiert. Insgesamt müssen $n * s$ Sektoren gelesen werden.

Es sollte ermittelt werden, ab welcher Blockgröße die maximal erzielbare Transferleistung der Festplatte erreicht wird. Dieser sequentielle Test wurde zwecks des Testens der JNI Schnittstelle identisch in reinem C implementiert.

Die maximal mögliche theoretische Datenrate der Festplatte beim Lesen des am weitesten außen liegenden Zylinders beträgt etwa 20 MB/s. Diese Zahl lässt

sich aus den im Datenblatt der Festplatte angegebenen Verzögerungszeiten berechnen. Hier sind jedoch keinerlei Verzögerungszeiten eingerechnet, die durch die Schnittstelle oder den Rechner selbst auftreten. Die theoretischen 20 MB/s können in der Praxis sicherlich nicht erreicht werden.

Die gemessenen Kurven sind in Abbildung 3.6 zu finden. Als erstes zeigt sich, dass sogar das sequentielle Lesen vergleichsweise langsam ist, wenn die Zugriffseinheiten klein sind. Bei einer Zugriffsgröße von einem Sektor pro Lesezugriff (512 Bytes) liegt die Datentransferrate gerade einmal bei 2 MB/s, d. h. bei einem Zehntel des theoretisch möglichen Wertes.

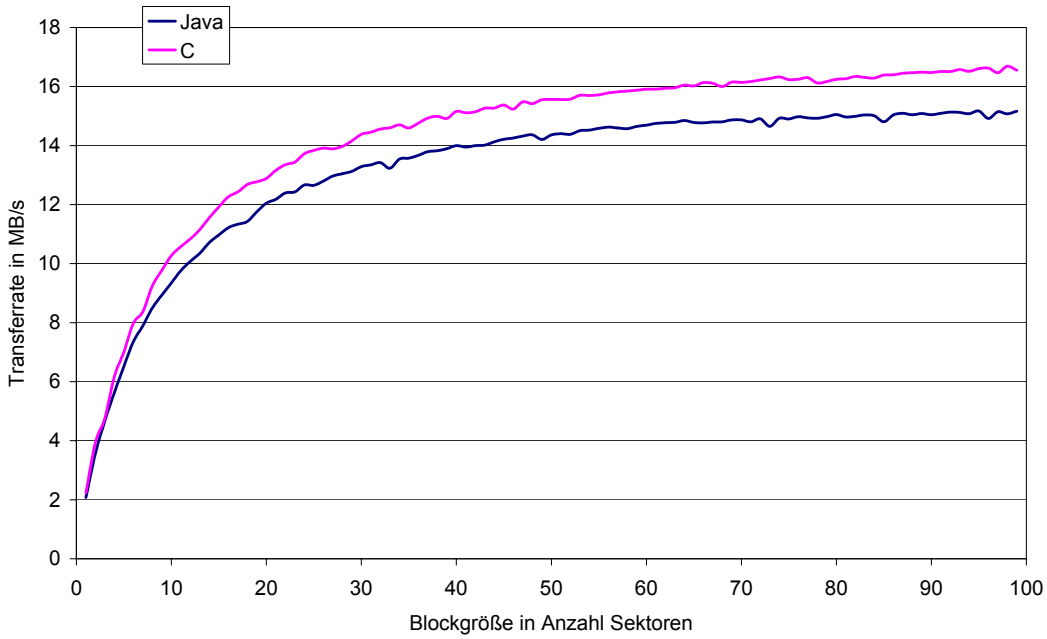


Abbildung 3.6: Messung der Datentransferleistung der IBM Festplatte bei sequentiellem Lesen

Transferraten von 15 MB/s sind in Java ab einer Zugriffsgröße von 70 Sektoren (35 KB) möglich, während die C-Implementierung noch 1,5 MB/s mehr Leistung bringt. Im Schnitt ist die C-Implementierung etwa 10% schneller. Dies ist nicht überraschend. Zum einen sind die JNI-Aufrufe ein Mehraufwand, der allerdings nicht sehr groß ist. Weiterhin läuft die *Garbage Collection* der Java Virtual Machine asynchron in einem eigenen *Thread*. Dies kostet einen kleinen Teil der möglichen Leistung. Zum dritten müssen bei der Java Implementierung die Bytefelder zwei Mal im Hauptspeicher hin und her kopiert werden.

Java schottet den direkten Zugriff auf die Bytefelder ab, so dass der Anwender nur eine Kopie anfordern kann (JNI-Funktion `GetByteArrayElements`) und diese anschließend zurückschreiben lassen muss (JNI-Funktion `Release-ByteArrayElements`). In der C-Implementierung sind solche Kopieraktionen nicht nötig.

Generell lässt sich sagen, dass die Leistung der in XXL implementierten nativen Festplattenanbindung bei sequentiellen Operationen zufriedenstellend ist. Java ist nun einmal im Schnitt einige Prozente langsamer als C und wer maximale Leistung benötigt, der muss seine Applikationen mit erhöhtem Entwicklungsaufwand in C implementieren.

Zugriffszeiten

Zur weiteren Untermauerung der Validität der Schnittstelle wurde ein Test implementiert, der die mittleren Zugriffszeiten bei Seeks verschiedener Längen misst. Bei diesem Test erfolgt zunächst eine Positionierung eines Festplattenkopfes auf Sektor 0. Anschließend wird Sektor n gelesen (mit ausgeschaltetem Festplattenpuffer) und die dafür benötigte Zeit der Seeklänge n zugeordnet.

Die Ergebnisse von mehreren Messungen wurden gemittelt. Die Kurve ist in Abbildung 3.7 zu finden. Sie zeigt die Charakteristika, die von Festplatten i. Allg. erwartet werden. Dies untermauert, dass die `RawAccess`-Schnittstelle in XXL sogar für solche Kurzzeitmessungen einsetzbar ist.

3.2.2.3 Anbindung an XXL

Um den direkten Festplattenzugriff in XXL nutzen zu können, ist das bisher Beschriebene nur unzureichend. Zwischen der höheren Funktionalität in XXL wie beispielsweise Indexstrukturen und dem direkten Festplattenzugriff klafft eine Lücke, die zu schließen war. An dieser Stelle wurden mehrere Wege beschritten.

Zunächst wurde ein Umsetzer (*Wrapper*) von `RawAccess` nach `java.io.RandomAccessFile` geschrieben. Hierdurch kann auf ein RawAccess genau so zugegriffen werden wie auf ein `RandomAccessFile`.

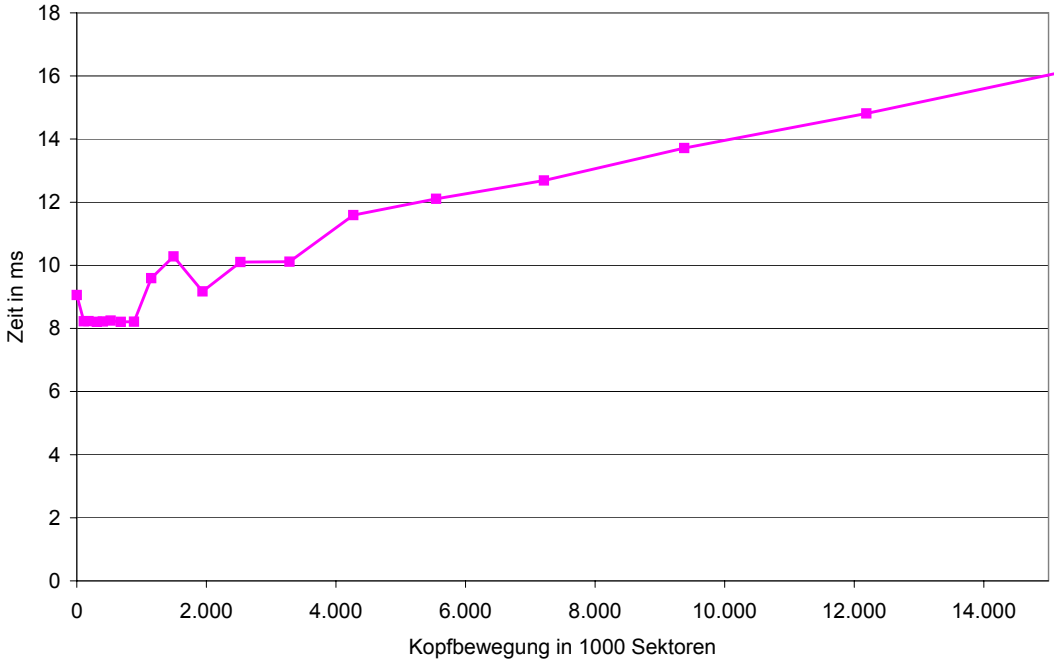


Abbildung 3.7: Messung der Zeit für Seeks verschiedener Länge bei der IBM Festplatte

Da die meisten Strukturen in XXL ihre Daten in Containern ablegen, musste es Container geben, die auf der `RawAccess`-Schnittstelle arbeiten. Ein `BlockFileContainer` legt seine Daten in fünf Dateien auf der Festplatte ab. Die erste Datei enthält die Blöcke selbst. Die vier anderen Dateien enthalten Verwaltungsinformationen, beispielsweise eine Bitliste die speichert, welche der Blöcke frei bzw. belegt oder reserviert sind. Die letztgenannten vier Dateien sind allesamt sehr klein, und ihre Größen können für eine vorgegebene maximale Anzahl an Blöcken einfach nach oben abgeschätzt werden.

Zur Abbildung des `BlockFileContainers` auf ein `RawAccess` wurde ein statischer Partitionierer implementiert, der ein `RawAccess` in mehrere Partitionen (hier: fünf) unterteilt. Diese sind einzeln wiederum über die `RawAccess`-Schnittstelle ansprechbar. Mit Hilfe des Umsetzers von `RawAccess` nach `RandomAccessFile` kann jede Partition als normale Datei in Java genutzt werden. Nach geringfügigen Modifikationen am `BlockFileContainer` selbst, kann nun ein `BlockFileContainer` sowohl mit Dateien des Betriebssystems, als auch mit direktem Festplattenzugriff verwendet werden.

Eine weitere Lösung des Verbindungsproblems ist die Implementierung eines Dateisystems, das auf der RawAccess-Schnittstelle basiert. Dadurch kann der `BlockFileContainer` ebenso auf einem `RawAccess` betrieben werden. In XXL wurde ein einfaches FAT-Dateisystem [Mic99, Mic92] implementiert. Die Optimierung des Quelltextes ist allerdings bislang nicht weit fortgeschritten. Somit ist dieses Dateisystem von der Leistung her nicht konkurrenzfähig und für Laufzeittests noch nicht zu gebrauchen.

Die letzte Anbindung wurde implementiert, nachdem sich gezeigt hatte, dass ein `BlockFileContainer` auf simulierten `RandomAccessFiles` Nachteile gegenüber einer direkten Implementierung eines Containers auf einem `RawAccess` besitzt. Ein `RandomAccessFile` muss, will es kompatibel zur Java-Implementierung sein, beispielsweise neue Blöcke am Dateiende initialisieren, bevor diese genutzt werden. Dies ist für einen Container nicht nötig. Deshalb wurde ein neuer Container namens `RawAccessContainer` implementiert, der direkt auf einem einzigen `RawAccess` arbeitet. Für alle Tests wurde daher dieser Container benutzt.

3.2.3 Der Recordmanager

Bislang existieren in XXL drei Container, die ihre Daten auf dem Externspeicher ablegen. Zum einen sind dies die bereits angesprochenen `BlockFileContainer` und `RawAccessContainer`, die Blöcke identischer Länge verwalten. Zum anderen gibt es den `MultiBlockContainer`, der Blöcke verwaltet, deren Größe ein ganzzahliges Vielfaches ($n \geq 1$) einer gegebenen Grundblockgröße belegen.

Wenn es um die Speicherung von Objekten beliebiger Länge geht, sind alle genannten Container nicht einsetzbar. Die Blockgröße B muss im Fall des `MultiBlockContainers` ein Vielfaches der Blockgröße des Externspeichers betragen, damit Effizienz gegeben ist. Da in der Praxis $B \geq 512$ gilt, ist vom Standpunkt der Speicherplatzausnutzung kein vorhandener XXL Container für Objekte mit $|O| << 512$ geeignet.

Für diesen Zweck wird ein so genannter *Recordmanager* benötigt, der mehrere *Records* in einem Block unterbringt. Die maximal mögliche Größe eines Records

maxRecordSize liegt geringfügig unter der verwendeten Blockgröße B, da in jedem Block zusätzlich zu den Records noch Verwaltungsinformationen wie beispielsweise die Anzahl der Records gespeichert werden müssen.

In der Literatur wurde bereits eine Vielfalt an Recordmanagern vorgeschlagen [Sho95], die mit verschiedenen Techniken arbeiten. Ein Recordmanager sollte für unsere Zwecke verschiedene Kriterien erfüllen:

- Keine Blackbox: Eine reine Blackbox Implementierung ist in jedem Fall unzureichend. Hier sind auftretende Effekte bei Messungen nicht nachvollziehbar.
- Weitgehende Parametrisierung: Ein Recordmanager ist für die Leistung gewisser Algorithmen von zentraler Bedeutung. Somit müssen möglichst viele und weitreichende Parametrisierungsschalter vorhanden sein.
- Nahtlose Integration in XXL (Objekt- und Blockpufferung mit verschiedenen Strategien)
- Nutzung von direktem Festplattenzugriff
- Gute Performanz

Aus verschiedenen Gründen schieden die vorhandenen Recordmanager aus. Somit war eine Neuimplementierung in XXL notwendig. Dabei wurden selbstverständlich die Lehren aus vorhandenen Recordmanagern beachtet.

In der XXL Architektur bot es sich an, den Recordmanager als `Container` zu implementieren, da ein `Container` genau die benötigte Funktionalität zusammenfasst. Die Freispeicherverwaltung auf Blockebene ist bereits im `BlockFileContainer` vorhanden. Diese Funktionalität soll genutzt und nicht neu implementiert werden.

Der Recordmanager in der Klasse `xxl.core.collections.container.RecordManager` implementiert somit die `Container`-Schnittstelle und bekommt bei der Konstruktion einen bereits vorhandenen Container übergeben. In diesem legt der die Blöcke ab. Dieser Container kann momentan sinnvoller Weise

ein `BlockFileContainer`, ein `RawAccessContainer` oder ein `MultiBlockContainer` sein. Es kann aber zu Testzwecken auch jeder andere Container verwendet werden, beispielsweise ein schneller Hauptspeichercontainer (`MapContainer`).

Es können auch zwei verschiedene Recordmanager auf ein und dem selben `Container` arbeiten. Dies ist syntaktisch möglich und soll aufgrund der erhöhten Flexibilität auch unterstützt werden. Aufgrund dessen ist es notwendig, dass der Recordmanager eine Liste aller Blöcke mitführt, die er im darunter liegenden `Container` belegt. Wie später noch gezeigt wird, ist es für die Verwaltung der Blöcke weiterhin wichtig, dass über die freien Bytes pro Block Buch geführt wird. Da eine Blockliste sowieso vorhanden ist, erfolgt in ihr zusätzlich die Speicherung des Belegungszustandes der Blöcke (2 Bytes). Bei der aktuellen Implementierung wird die Blockliste mit dem Belegungszustand im Hauptspeicher gehalten und erst beim Schließen des Recordmanagers auf den Externspeicher geschrieben. In Zukunft könnte die Liste externspeichertauglich gemacht werden, damit nicht die komplette Liste zu jeder Zeit im Hauptspeicher gehalten werden muss, sondern nur Teile von ihr.

Beim Entwurf eines Recordmanagers sind zwei wichtige Entscheidungen zu treffen:

- Das Identifikatorkonzept: Wie wird ein Record eindeutig wiedergefunden?
- Die Platzierungsstrategie: Wie wird bestimmt, in welchen Block ein Record beim Einfügen bzw. bei Vergrößerung eingefügt wird.

Diese beiden orthogonalen Fragen werden in den folgenden Unterkapiteln diskutiert. Anschließend erfolgt eine kurze Evaluierung des erstellten Recordmanagers.

3.2.3.1 Identifikatoren

Die physische Position eines Records innerhalb eines Recordmanagers kann eindeutig durch einen *physischen Tupelidentifikator* beschrieben werden, der aus

einer Blocknummer und einem Identifikator innerhalb des Blocks besteht. Dies birgt jedoch Probleme. Wenn ein Record vergrößert wird, kann es passieren, dass es irgendwann nicht mehr in den ursprünglichen Block hineinpasst und in einen anderen Block mit mehr freiem Speicher verschoben wird. Der ursprüngliche Identifikator verweist dann nicht mehr auf das Record, auf das er zeigen sollte. Somit wäre jeder Identifikator nur ein Identifikator auf Zeit, der bei einem Blockwechsel des Records seine Gültigkeit verliert.

In einem System sind häufig viele Referenzen auf ein und dasselbe Record vorhanden. In Indexstrukturen stehen Referenzen sogar persistent auf dem Externspeicher. An anderer Stelle werden sie über das Netzwerk an andere Rechner übermittelt und dort gespeichert. Wenn ein Record verschoben wird, so müssten alle diese Identifikatoren auf den neuen Stand gebracht werden. Dies wäre extrem aufwendig, da über jeden Identifikator genau Buch geführt werden müsste. In der Praxis ist so etwas ineffizient und auch gar nicht mit vertretbarem Aufwand realisierbar.

Die grundlegende Bedingung an ein Konzept für Identifikatoren lautet demnach: Der Recordmanager muss mit jedem Identifikator, den er ein Mal nach außen gegeben hat, ein vorhandenes Record wiederfinden können. Dazu gibt es zwei substantiell verschiedene Konzepte [HR01].

Das erste Konzept, das so genannte Abbildungskonzept, verwaltet eine Abbildung von Identifikatoren beliebigen Typs auf physische Tupelidentifikatoren. Beispielsweise wird ein Ganzzahl-Datentyp wie Long (8 Bytes in Java) als Recordidentifikator verwendet. Er wird auf einen physischen Tupelidentifikator bestehend aus einem Long (Identifikator für einen Externspeicherblock) und einem Short (2 Bytes in Java, Identifikator für ein Record innerhalb des Externspeicherblocks) abgebildet. Da nun jeder Zugriff über diese Abbildung läuft, muss diese effizient berechenbar sein, d. h. im Wesentlichen ohne Externspeicherzugriffe. Allerdings kann die Abbildung sehr groß werden, da für jedes Record der logische und der physische Identifikator, sowie eventuell Zusatzinformationen für die Abbildung gespeichert werden müssen.

Das zweite Konzept, das Verweiskonzept, basiert stark auf den bekannten physischen Tupelidentifikatoren, erweitert diese jedoch um das Konzept von Ver-

weisen. Wenn ein Record in einen anderen Block verschoben wird, so wird an der ursprünglichen Stelle statt dem Record ein physischer Tupelidentifikator mit der neuen physischen Position des Records eingefügt. Beim Zugriff mittels des alten Tupelidentifikators erkennt der Recordmanager die dort liegenden Daten als Verweis[3] und geht zu den eigentlichen Daten des Records weiter. Zu beachten ist, dass niemals längere Ketten von Verweisen entstehen können. Wenn ein Record ein zweites Mal verschoben werden sollte, so wird der ursprüngliche Verweis auf die neue Position abgeändert. Wichtig ist, dass niemals ein Identifikator auf die neue Position nach außen gegeben wird. Wird diese Regel nicht beachtet, entstehen potentiell längere Ketten. Dies liegt daran, dass nicht effizient ermittelt werden kann, wo ein zu verändernder Verweis auf den herausgegebenen Verweis überall vorhanden ist. Beim zweiten Verschieben des Records hätte somit der ursprüngliche physische Recordidentifikator zwei Verweisen auf dem Weg zu den Daten des Records zu folgen.

Der Nachteil des zweiten Konzepts besteht darin, dass im Fall von vielen Recordvergrößerungen einiges an Speicher für die Verweise benötigt wird (besonders bei vielen kleinen Records). Außerdem kann es dann pro Zugriff auf ein Record zu zwei E/A Zugriffen kommen.

Im Recordmanager von XXL wurden beide Konzepte realisiert. Es wurde eine Schnittstelle definiert, die Methoden für alle wichtigen, die Identifikatoren betreffende Vorgänge enthält (`xxl.core.collections.containers.recordManager.TIdManager`). Ein TIdManager wandelt logische Identifikatoren in physische um und umgekehrt. Er wird informiert, wenn Records verschoben wurden und wenn gewisse Identifikatoren entsorgt werden können. Außerdem sagt der TIdManager dem Recordmanager, ob dieser mit Verweisen arbeiten soll oder nicht.

Das Abbildungskonzept wurde in der Klasse `MapTIdManager` realisiert, welche die `TIdManager`-Schnittstelle implementiert. Hierbei wurde `java.util.Map` für die Abbildung verwendet. Das Konzept mit den Verweisen ist in der Klasse `IdentityTIdManager` zu finden. Die Implementierung ist hier sehr einfach, weil Identifikatoren nicht verändert werden müssen und dem Recordmanager

[3] Hierzu ist in jedem Block ein kleines, in der Größe vernachlässigbares Bitfeld notwendig.

nur gesagt werden muss, dass er mit Verweisen arbeiten soll. In Zukunft soll eine dritte Variante hinzukommen, die das Abbildungskonzept realisiert, die dafür benötigten Daten allerdings auf dem Externspeicher ablegt. Für Effizienz soll hier ein großzügig bemessener Puffer sorgen.

3.2.3.2 Platzierungsstrategien

Wichtiger als die Wahl der Strategie für die Identifikatoren ist für die Leistung des Recordmanagers die Wahl der Platzierungsstrategie. Die Platzierungsstrategie legt fest, in welchen Block ein Record einer bestimmten Größe eingefügt wird. Ein Recordmanager besitzt hierbei extrem viele Möglichkeiten.

Allgemein wird eine möglichst gute Lösung für dieses (*Online Object Placement Problem* [MCS96]) gesucht. Dieses Problem ähnelt stark dem Rucksackproblem, bei dem eine vorgegebene Menge an Gegenständen möglichst platzsparend zusammengepackt werden soll. Allerdings ist das Problem hier von inkrementeller Natur („Online“) und eine Entscheidung ist immer nur für den nächsten Gegenstand, also hier für das nächste Record, zu treffen. Eine einmal getroffene Entscheidung wird nicht mehr korrigiert. Eine nachträgliche Verschiebung von Records wäre zwar prinzipiell möglich, wäre allerdings sehr komplex und wurde in der Forschung bislang nicht betrachtet.

Für XXL wurde eine Schnittstelle für Platzierungsstrategien entwickelt. Die wichtigste Methode muss einen passenden Block für ein Record bestimmter Größe zurückgeben. Über weitere Methoden der Schnittstelle wird die Strategie über Veränderungen aller Blöcke auf dem Laufenden gehalten. Beispielsweise erfährt die Strategie, wenn Blöcke neu allokiert wurden oder wenn Änderungen des freien Speichers innerhalb vorhandener Blöcke auftraten.

3.2.3.3 Platzierungsstrategien aus der Literatur

In diesem Abschnitt sollen einige Platzierungsstrategien vorgestellt werden, die aus der Literatur stammen und im Recordmanager von XXL implementiert wurden. Der grundlegende Artikel von McAuliffe, Carey und Solomon [MCS96] hat seine Platzierungsstrategien im objektorientierten Datenbanksystem Shore realisiert und mit selbigem evaluiert.

Eine weitere Strategie wurde mitsamt der ihr zugrunde liegenden zwei Strategien aus [Kan03] entnommen. Für diese drei Strategien wurden jedoch keine Evaluierungen durchgeführt, weder in der angegebenen Quelle, noch in der uns bekannten Literatur.

Im folgenden werden alle Strategien kurz beschrieben. Für weitere Details der Verfahren wird an dieser Stelle auf die Dokumentation der Implementierung, sowie auf die angegebene Literatur verwiesen.

Vorab sei bereits auf eine Gemeinsamkeit der Strategien hingewiesen: Wenn ein Record in keinen der vom Recordmanager verwalteten Blöcke passt, so wird ein neuer Block allokiert und das Record in diesen eingefügt. Diese Gemeinsamkeit wird somit im Folgenden nicht weiter aufgeführt.

First Fit (FF) [MCS96] Es werden sequentiell die Blöcke des Recordmanagers durchsucht, und zwar beginnend mit dem Anfang (dem ersten Block des Recordmanagers). Es wird der erste Block verwendet, der genügend freien Speicher für das Record besitzt.

Next Fit (NF) [MCS96] Es wird sequentiell wie bei First Fit gesucht, allerdings nicht beginnend beim ersten Block, sondern beim dem Block, wo zuletzt ein Record eingefügt oder gelöscht wurde.

Best Fit (BF) [MCS96] Es wird wie bei First Fit gesucht. Ein Block wird jedoch nur dann direkt verwendet, wenn der freie Speicher in ihm nach der Reservierung unterhalb einer gewissen vorgegebenen Grenze liegt. Wird kein Block auf diese Weise gefunden, so wird derjenige Block genommen, in dem nach der Reservierung möglichst wenig freier Speicher verbleibt.

Append Only (AO) [MCS96] Es wird immer nur anhängend eingefügt, das heißt nur in den zuletzt allokierten Block.

Append Only n (AO(n)) [MCS96] Es wird im Wesentlichen anhängend eingefügt. Allerdings darf hierbei in die n zuletzt allokierten Blöcke eingefügt werden. Hierbei hat der älteste der n Blöcke Priorität.

Next Fit mit Histogramm (NFH(n)) [MCS96] Es wird prinzipiell mit Next Fit gesucht. Zunächst erfolgt jedoch eine Abschätzung, ob ein passender Block innerhalb der bereits vorhandenen Blöcke vorhanden sein muss. Hierzu wird ein äquidistantes Histogramm verwendet.

Jeder Histogrammklasse ist ein Intervall zugeordnet, dessen Randwerte einem Prozentsatz an freiem Speicher entsprechen. Die untere Intervallgrenze ist inklusive des Randwerts und die obere Grenze exklusive des Randwerts (bei der letzten Klasse inklusive). Für jede Histogrammklasse wird gespeichert, wie viele Blöcke mit ihrem freien Speicherplatz in sie fallen.

Dieses Histogramm kann durch die Methoden der Strategieschnittstelle aktuell gehalten werden. Bei einer Anfrage, in welchen Block ein Record eingefügt werden soll, wird zunächst überprüft, ob ein Record mit genügend freiem Speicherplatz existiert. Wenn dieses existiert, so wird wie gehabt mit Next Fit gesucht, ansonsten wird die teure, sehr wahrscheinlich erfolglose Suche nicht durchgeführt und das Record in einen neuen, leeren Block eingefügt.

Next Fit mit Histogramm und Stellvertretern (NFHS(n)) [MCS96] Diese Strategie basiert, wie der Name schon andeutet, auf der Next Fit mit Histogramm Technik. Zusätzlich zu jeder Histogrammklasse wird der Identifikator von einem Stellvertreterblock gespeichert. Der referenzierte Block wird benutzt, wenn eine Anfrage mit einem Block der Histogrammklasse beantwortet werden kann. Wenn in einer Histogrammklasse kein Stellvertreter vorhanden ist, so wird dieser beim Besuch eines passenden Blocks während einer Suche wieder aufgefüllt. Das Auffüllen bedeutet also einen zusätzlichen Aufwand.

Hybrides Verfahren (HY(n,u)) [MCS96] Dieses Verfahren setzt auf leicht modifizierten Varianten der Strategien AO(n) und NFH(n) auf und kombiniert diese. Der Parameter u beschreibt die gewünschte, mittlere Speicherplatzausnutzung aller Blöcke des Recordmanagers. Es sei nun v die tatsächlich vorhandene mittlere Speicherplatzausnutzung. Der Wert von v kann über das sowieso vorhandene Histogramm ausreichend abgeschätzt oder auch exakt mit Hilfe des Recordmanagers berechnet werden. Es gibt nun zwei Fälle:

- Wenn $v > u$, so wird ein modifiziertes AO(n) durchgeführt.
- Wenn $v \leq u$, so wird ein modifiziertes NFH(n) durchgeführt.

Die modifizierten Verfahren AO(n) und NFH(n) unterscheiden sich von den oben beschriebenen Verfahren dadurch, dass Blöcke nur dann in der Menge M der n gespeicherten Blöcke landen, wenn sie eine Speicherplatzausnutzung kleiner als u besitzen (also mehr freien Speicher als der Schnitt aller Blöcke des Recordmanagers). Weiterhin wird bei der Suche in NFH(n) ein Block B nur dann genommen, wenn es in M einen Block mit weniger freiem Speicher gibt. In diesem Fall wird der Block mit der höchsten Belegung aus M durch B ersetzt.

Last Fit (LF) [Kan03] Diese Strategie ist komplementär zu First Fit. Hier wird ausgehend vom letzten Block des Recordmanagers zurück bis zum ersten Block gesucht. Da neu angeforderte Blöcke zunächst am Ende des Recordmanagers erscheinen, kann die nächste Suche dann häufig nach nur einem einzigen oder nach wenigen Tests beantwortet werden. Dies ist ein Vorteil im Vergleich zur First Fit Strategie.

In der angegebenen Quelle wurde diese Strategie als Basis für die im Folgenden beschriebene LFLRU(n)-Strategie verwendet. Eine Evaluierung erfolgte wie bereits angesprochen weder dort, noch an anderen Stellen in der mir bekannten Literatur.

Least Recently Used (LRU(n)) [Kan03] Diese Strategie versucht, möglichst gut mit einem LRU-Blockpuffer zusammen zu arbeiten und Records immer in Blöcken unterzubringen, die noch im Puffer liegen. Dadurch sollen Externspeicherzugriffe vermieden werden. Es wird intern eine Liste der zuletzt zugegriffenen n Blöcke aktuell gehalten.

Diese Strategie ist die zweite Basisstrategie für die im folgenden beschriebene LFLRU(n)-Strategie. Eine Evaluierung von LRU(n) ist ebenso nicht verfügbar, wie dies zuvor bei der LF-Strategie schon der Fall war.

Hybrides LRU mit Last Fit (LRULF(n)) [Kan03] LRULF(n) ist eine hybride Strategie, die zunächst LRU(n) benutzt und, wenn hierdurch kein Block für ein Record gefunden wird, auf die LF-Strategie umschaltet. Dieses Verfahren wurde in [Kan03] für alle Tests von den dort entwickelten Verfahren zur Speicherung von XML benutzt. Jedoch erfolgte dort keine Evaluierung der Recordmanagerstrategie, wodurch sich die Ergebnisse der Studie auch schlecht einordnen lassen.

3.2.3.4 Neu entwickelte Platzierungsstrategien

Neben diesen Strategien aus der Literatur wurden vier weitere Strategien entwickelt und implementiert. Sie basieren auf eigenen Ideen und werden im Folgenden vorgestellt.

Best Fit on n emptiest blocks (BFE(n)) Diese Strategie speichert die Identifikatoren zu n Blöcken, die möglichst viel freien Speicher besitzen. Aus diesen n Blöcken wird bei einer Anfrage ein Block mittels der Best Fit Strategie gesucht.

Die n Blöcke werden in einer nach dem freien Speicherplatz sortierten Liste verwaltet. Wenn ein Block bei einer Operation betrachtet wird, so wird anschließend überprüft, ob er mehr Speicher frei hat als der vollste der n Blöcke. Ist dies der Fall, so wird er mit diesem in der Liste ausgetauscht.

Für den Fall, dass keine Einfügeoperationen auftreten, sind die n referenzierten Blöcke immer die n leersten Blöcke des gesamten Recordmanagers. Bei Einfügeoperationen tritt jedoch das Problem auf, dass häufig einer der n Blöcke so weit wächst, dass er zum vollsten der n Blöcke wird. An dieser Stelle müsste, wenn wirklich immer die n leersten Blöcke in der Liste gehalten werden sollen, der komplette Recordmanager durchsucht werden, um den n-leersten Block zu finden. Dieser würde dann in der Liste an die Stelle des gerade gewachsenen, vollsten Blocks treten. Da diese Suchoperation sehr teuer ist, wird auf sie verzichtet. Trotzdem erreicht das Verfahren bei normalen Anfragelasten, dass die gespeicherten n Blöcke nahe an den n leersten Blöcken sind.

Diese Platzierungsstrategie ist deshalb erfolgsversprechend, weil die n Blöcke normaler Weise viel Platz für neue Records bieten und deshalb die Suche nach einem freien Block sehr schnell und häufig erfolgreich beantwortet werden kann. Blöcke mit hohem Belegungsgrad müssen bei dieser Strategie überhaupt nicht betrachtet werden, wodurch Rechenzeit eingespart wird.

Hybrides BFE(n) mit NFH(n) (BFENFH(n)) BFENFH(n) ist eine hybride Strategie, die auf BFE(n) basiert. Wenn die BFE(n) Strategie einen neuen Block anlegen müsste, weil das Record in keinen vorhanden Block des Recordmanagers passt, so wird hier jedoch die NFH(n) Strategie nach einer besseren Platzierungsmöglichkeit gefragt.

Diese Strategie beseitigt das Manko von BFE(n), wenn n für eine Anfragelast zu klein gewählt wurde. Wenn viele Einfügeoperationen in unmittelbarer Folge auftreten, so sind die n Blöcke unter Umständen relativ bald voll und die Strategie hat nur die Möglichkeit, neue Blöcke anzulegen. An dieser Stelle springt die NFH(n)-Strategie ein und sorgt dafür, dass auch andere Blöcke betrachtet werden und die Speicherplatzausnutzung hoch bleibt.

Hybrides BFE(n) mit NFHS(n) (BFENFHS(n)) Diese Strategie ist analog zur BFENFH(n)-Strategie, nur dass hier die zweite Strategie nicht NFH(n), sondern NFHS(n) ist.

One record per block (1RPB) Triviale Strategie, bei der immer nur ein Record pro Block gespeichert wird. Diese Strategie dient im Wesentlichen dem Vergleich mit der Situation, wenn gar kein Recordmanager eingesetzt wird, sondern auf einem blockbasierten Container gearbeitet wird.

3.2.3.5 Theoretische Beurteilung der Platzierungsstrategien

Die wichtigsten Kriterien für die Beurteilung von Strategien für Recordmanager sind Hauptspeicherverbrauch, Rechenzeit, Speicherplatzausnutzung (SPAN) und das Zugriffsverhalten auf dem Externspeicher. Alle Strategien haben gemeinsam, dass der Hauptspeicherverbrauch generell sehr gering ist, da der Pa-

rameter n für ein akzeptables Verhalten normaler Weise nicht sehr groß gewählt werden muss ($n < 50$). Somit wird auf eine genauere Diskussion dieses Punktes verzichtet.

Zunächst werden die Verfahren anhand der genannten Kriterien theoretisch untersucht. In Abschnitt 3.2.3.6 erfolgt dann eine praktische Evaluierung mit vorgegebenen Daten und Anfragelasten.

Rechenaufwand Den höchsten Rechenaufwand besitzt die BF-Strategie, da bei ihr für jedes neu eingefügte Record der freie Speicher für alle Blöcke betrachtet werden muss. Erst dann kann eine Entscheidung getroffen werden, es sei denn, das Record passt exakt in den freien Speicher eines Blocks. Bei n Einfügeoperationen in einen leeren Recordmanager beträgt der Aufwand demnach $O(n^2)$. Die hier benutzte Variante von BF fügt ein Record bereits dann in einen Block ein, wenn die Speicherplatzausnutzung des Blocks oberhalb einer vordefinierten Schranke liegt. Dadurch wird der Rechenaufwand reduziert. In den Tests wurde immer ein Wert von 5% verwendet. Auf den praktischen Test von BF(0%) wurde aufgrund des inakzeptabelen Rechenaufwandes verzichtet.

FF, NF und LF besitzen in der Realität einen geringfügig niedrigeren tatsächlichen Aufwand als BF, was sich in der O-Notation jedoch nicht niederschlägt. Das Hauptproblem von den Verfahren ist, dass eine erfolglose Suche immer alle Blöcke betrachten muss. Dies wird durch Histogrammverfahren deutlich gemildert. Durch die Befragung eines Histogramms wird sichergestellt, dass die Suche nach freiem Speicherplatz in jedem Fall erfolgreich verläuft[4]. Das NF-Verfahren mit Stellvertreter optimiert dies weiter. Allerdings belegt die zusätzliche Liste Speicherplatz, der bei den anderen Verfahren nicht benötigt wird.

AO und 1RPB sind von der Rechenzeit her die schnellsten Verfahren. AO(n), HY(n,u), LRU(n) und BPE(n) sind ebenfalls sehr schnell, kommen aber an AO und 1RPB nicht heran. Die Leistung von LFLRU(n) hängt wesentlich von der Ausnutzung des n-elementigen Puffers ab. Ist er für eine gegebene Anfragelast zu klein, so leidet das Verfahren sehr unter dem langsamen LF-Anteil. Die auf

[4]Es ist möglich, dass eine Suche erfolgreich verlaufen würde, sie jedoch nach Befragung des Histogramms trotzdem nicht durchgeführt wird.

BPE(n) basierten Strategien mit NFH(n) bzw. NFHS(n) müssen zwei Datenstrukturen aktuell halten, weshalb der CPU-Aufwand etwas höher liegt, als bei BPE(n). Genaueres muss hier die praktische Evaluierung zeigen.

Speicherplatzausnutzung Bei der Speicherplatzausnutzung muss die 1RPB Strategie natürlich ganz hinten liegen. AO und AO(n) haben genau dann eine schlechte SPAN, wenn während dem Lauf einer Applikation viele Records verändert oder ganz gelöscht wurden. LRU(n) wird ebenfalls keine gute SPAN aufweisen, da die n zuletzt zugegriffenen Seiten nicht unbedingt viel freien Speicher besitzen müssen. Dies trifft insbesondere dann zu, wenn viele Leseoperationen zwischen den Schreiboperationen auftreten.

FF, NF und LF besitzen auf jeden Fall eine deutlich bessere SPAN als die vorher genannten Verfahren. Bei FF besitzen die ersten Blöcke des Recordmanagers i. d. R. einen sehr hohen Füllgrad. BF sollte immer den höchsten Füllgrad aller Verfahren aufweisen, selbst wenn das 5%ige Abbruchkriterium gewählt wird.

Wo sich LRULF(n), BPE(n), BPENFH(n) und BPENFHS(n) einordnen, müssen Experimente zeigen. Theoretische Aussagen sind hier schwierig und hängen außerdem stark vom gewählten n ab.

Die HY(n,u) Strategie hat neben n noch den Parameter u, der einen großen Einfluss besitzt. u sollte in der Nähe der real, in einer konkreten Applikation möglichen Speicherplatzausnutzung liegen. Ist u zu hoch gewählt, so degeneriert das Verfahren zu einem langsamen NFH(n). Ist u zu niedrig, so wird Speicherplatz verschenkt. Die Untersuchung von Verfahren mit einer dynamischen Anpassung des Parameters u wären sicherlich in Zukunft interessant.

Zugriffsverhalten auf dem Externspeicher In realen Applikationen kommt es zu weiteren Effekten, die durch den Recordmanager und seine benutzte Strategie induziert sind. An dieser Stelle sollen drei Szenarien und ihre Folgen für den Recordmanager kurz angesprochen werden.

Im ersten Szenario speichert eine Applikation eine Menge von Records und liest diese anschließend mit hoher Wahrscheinlichkeit auch wieder zusammen

aus (zeitliche Lokalität). Hierfür sind Platzierungsstrategien wie 1RBP, AO, FF oder NF zu empfehlen, die dafür sorgen, dass zeitlich nacheinander eingefügte Records im selben oder in nahe beieinander liegenden Blöcken gespeichert werden (räumliche Lokalität). Dies verringert die Anzahl der Seeks.

Im zweiten Szenario sind die Zugriffe gleichverteilt. Nahezu jeder Zugriff führt unweigerlich zu einem Seek. Hier sind Verfahren vorzuziehen, die eine hohe Speicherplatzausnutzung bieten, so dass der Recordmanager weniger Blöcke belegt. Puffer werden hierdurch besser genutzt. Das Argument der Pufferausnutzung spricht für LRU(n), da diese Strategie versucht, Einfüge- bzw. Änderungsoperationen auf die momentan gepufferten Seiten zu beschränken.

Das dritte und letzte Szenario sieht so aus, dass eine Applikation in kurzer Folge sehr viele Records einfügen will. Hier sollte zum einen ein Verfahren eingesetzt werden, das viel sequentielle Zugriffe erzeugt. Zum anderen kann sich hier ein quadratischer Rechenaufwand für die Platzierungsstrategie entscheidend negativ auswirken. Durch den Rechenaufwand scheiden in diesem Szenario FF, NF, LF und BF aus.

Über die hybriden Verfahren kann zusammenfassend gesagt werden, dass sie sowohl Seeks produzieren, als auch einen niedrigen Rechenaufwand und eine gute SPAN besitzen. Eine genaue Einordnung erfolgt im nächsten Kapitel.

3.2.3.6 Praktische Evaluierung der Platzierungsstrategien

In diesem Abschnitt geht es um die praktische Evaluierung des implementierten Recordmanagers. Dies erfolgt anhand verschiedener Testszenarien, die in Tabelle 3.1 aufgeführt sind. Die Szenarien sind deutlich voneinander verschieden, decken jedoch die Einsatzszenarien für einen Recordmanager noch nicht komplett ab. Für eine vollständige Evaluierung wären sicherlich weitere Szenarien nötig. In den folgenden Kapiteln wird daher die Evaluierung fortgesetzt, indem die dort erzeugten Anfragelasten zum Test des Recordmanagers eingesetzt werden. Diese sind im Vergleich zu den synthetischen Anfragelasten sehr viel praxisnäher.

Szenario	Beschreibung
RM1	Führe i-Mal aus: 1. Füge Records ein bis r Records vorhanden sind. 2. Wähle $r/2$ Records zufällig aus und verändere diese. 3. Außer beim letzten Durchlauf: Wähle $r/2$ Records zufällig aus und lösche sie.
RM2	Füge möglichst schnell r Records ein.
RM3	Füge r Records ein und führe i-Mal Änderungsoperationen auf $r/2$ zufällig ausgewählten Records durch.
RM4	Füge r Records ein. Führe dann i-Mal r Operationen durch. Von den r Operationen sind u Prozent zufällige Änderungsoperationen und der Rest Einfügeoperationen. Änderungsoperationen und Einfügeoperationen werden zufällig gemischt ausgeführt.

Tabelle 3.1: Szenarien zum Testen des Recordmanagers

In den nun folgenden Tests besitzen die Records einen zufälligen Inhalt. Die Recordgrößen sind zufällig um 128 Bytes verteilt, was einem Viertel der minimalen Blockgröße von 512 Bytes entspricht[5].

Der Recordmanager wurde mit den verschiedenen Identifikatorkonzepten und Platzierungsstrategien auf Basis eines `RawAccessContainers` betrieben. Bei Tests mit Puffer wurde zwischen `RecordManager` und `RawAccessContainer` ein `BufferedContainer` mit einem `LRUBuffer` geschaltet. Die Kapazität des LRU Puffers betrug 10% der Größe der Datenmenge. Der `RawAccessContainer` benutzt seinerseits ein `RawAccess`, das ausschließlich den Hauptspeicher verwendet und die E/A-Zugriffe nur zählt. Da alle Operationen bei diesem Versuchsaufbau im Hauptspeicher ablaufen, ist die gemessene Laufzeit ein gutes Maß für die Prozessorbelastung durch das gemessene Verfahren. Neben der CPU-Last wurde noch die Speicherplatzausnutzung (SPAN) innerhalb des Recordmanagers gemessen.

[5]Daraus folgt eine Gleichverteilung der Recordgröße zwischen einer minimalen Größe min und der halben minimalen Blockgröße minus min.

Die Parameter, die während der Experimente variiert wurden, sind in Tabelle 3.2 zu finden. Festgesetzt wurden: $r = 1000, i = 50, u = 0.85$. Bei RM2 wurde generell $r = 10000$ gewählt.

Parameter	Beschreibung
Blockgröße B	Die Größe der Blöcke in Bytes: 512, 2048, 8192
Puffergröße P	Blockpuffer: 0% oder 10% der Datenmenge. Bei RM2 wird mit 1% statt 10% gearbeitet. Es wurden Write-Through und Write-Back Strategien getestet.
Identifikatoren	Verweiskonzept oder Abbildungskonzept
Platzierungsstrategie	Alle 15 implementierten Strategien: 1RPB, FF, NF, BF, AO, AO(n), NFH(n), NFHS(n), LF, LRU(n), LRULF(n), HY(n,u), BFE(n), BFENFH(n), BFENFHS(n)
n	Parameter n der Platzierungsstrategien. Gewählte Werte: 5, 10, 20
p_a	Parameter für Szenario RM4, der den Prozentsatz der Änderungsoperationen im Verhältnis zur Gesamtzahl der Operationen (Einfüge- und Änderungsoperationen) festlegt.

Tabelle 3.2: Die Parameter beim Test des Recordmanagers

Insgesamt wurden über 10000 Einzelexperimente durchgeführt. Die dadurch erzeugten Daten wurden mit einer Tabellenkalkulation analysiert. Die wichtigsten Resultate werden im Folgenden vorgestellt. Die kompletten Messdaten werden auf Anfrage zur Verfügung gestellt.

Generell wird auf die Darstellung der Auswirkung des Write-Through Puffers verzichtet, da sich die Write-Back Pufferung in allen Experimenten als überlegen dargestellt hat. Zunächst folgen Ergebnisse in den genannten Kriterien, die durch die Analyse der Szenarien RM1 bis RM3 erzielt wurden. Daran anschließend werden für ausgewählte Platzierungsstrategien Szenario RM4 getestet.

Rechenaufwand Der Rechenaufwand ist bei Szenario RM1 bei allen Verfahren relativ ähnlich und niedrig (siehe Abbildung 3.8). Lediglich BF und FF

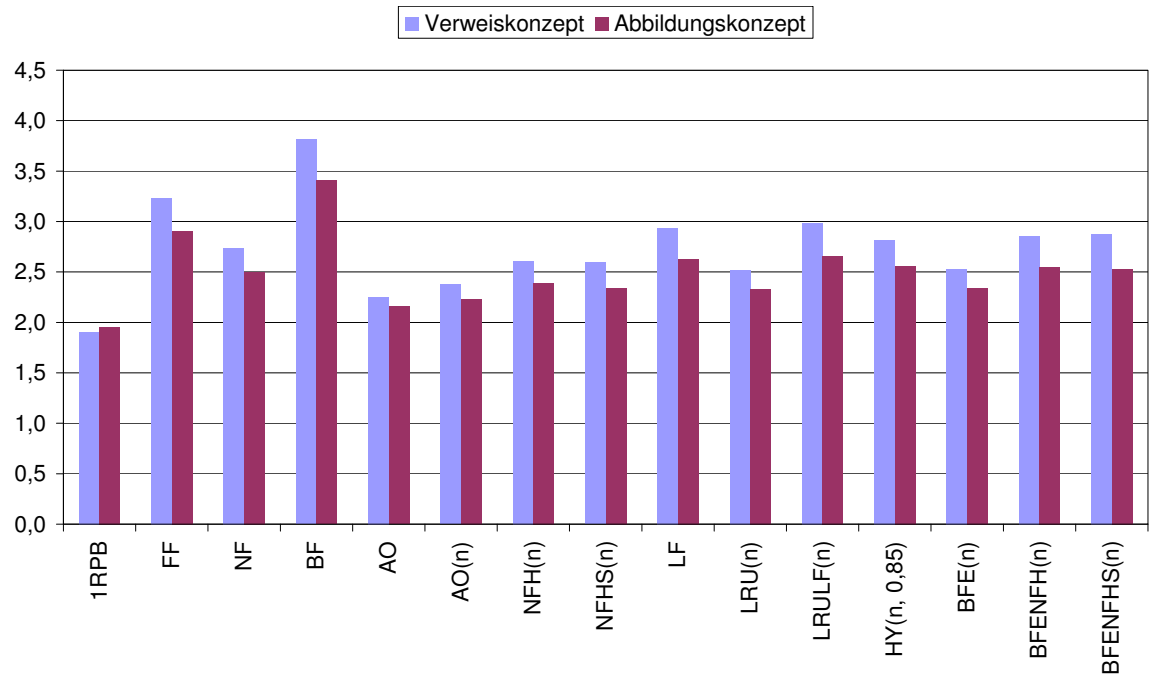


Abbildung 3.8: Rechenzeit (in Sekunden) für verschiedene Identifikatorkonzepte und Platzierungsstrategien. RM1, B=512, P=0, n=10

fallen hier mit 99,9% bzw. 69,6% Mehraufwand gegenüber dem schnellsten Verfahren 1RPC deutlicher ab. In Anbetracht dessen, dass in diesem Szenario der E/A-Aufwand die dominierenden Kosten verursacht, sind jedoch selbst diese Ausreißer hier unkritisch und sprechen nicht gänzlich gegen den Einsatz der Verfahren. Ein ähnliches Bild bietet Szenario RM3, wobei die Abstände zwischen den Verfahren noch enger sind. Auf eine Grafik wurde hier aufgrund der geringen Mehraussage verzichtet.

Deutlicher wird der Unterschied bei Szenario RM2 (siehe Abbildung 3.9). Das Szenario RM2 tritt in der Praxis immer dann auf, wenn größere Datenmengen im Pulk eingefügt werden, was eine relativ häufige Operation ist. Hier ist AO das beste Verfahren. Die quadratischen Verfahren FF, NF, LF und BF sind um Faktoren von bis zu 108 langsamer als AO. Hierdurch dominiert die Rechenlast bei diesen Experimenten die Zeit für Externspeicherzugriffe. Somit sind die Verfahren FF, NF, LF und BF nicht praxistauglich. Auch LRULF(n) scheidet hier aus, da der LF-Anteil im Szenario RM2 überwiegt. Bei größeren Blockgrößen sinkt allerdings der Abstand der Verfahren wieder, weil bei hal-

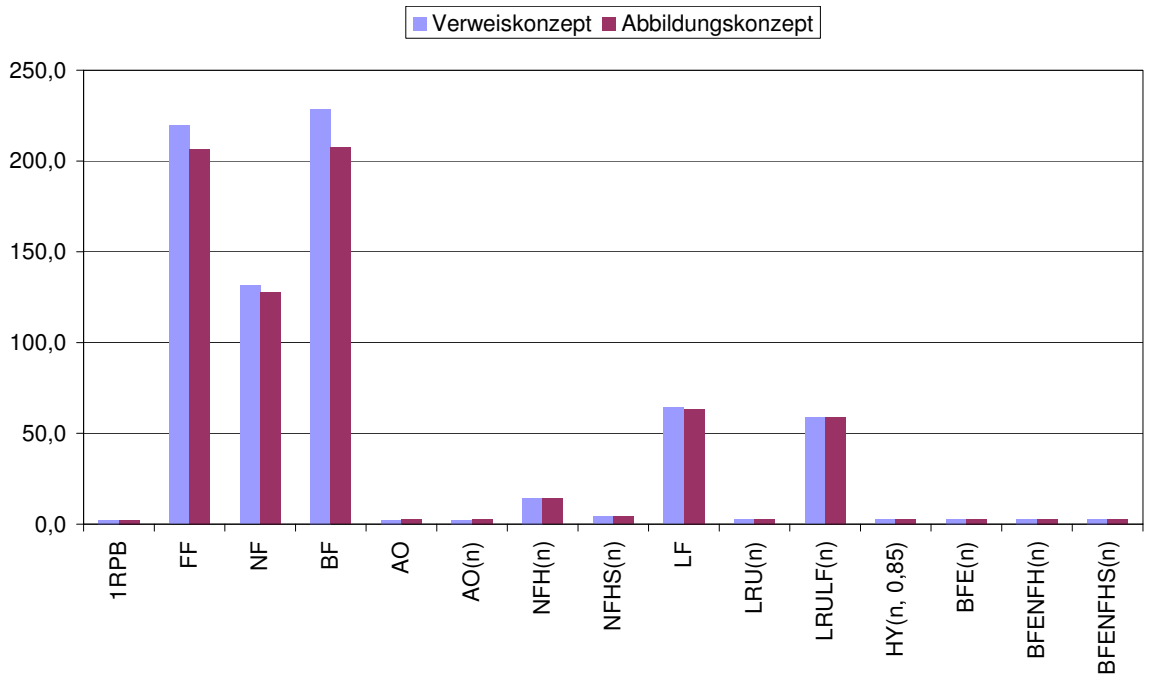


Abbildung 3.9: Rechenzeit (in Sekunden) für verschiedene Identifikatorkonzepte und Platzierungsstrategien. RM2, B=512, P=0, n=10

ber Blockgröße bei einem quadratischen Verfahren nur ein Viertel der Seiten untersucht werden muss.

In den bisher vorgestellten Messungen ist das Identifikatorkonzept, das mit der Hauptspeicherabbildung arbeitet, immer etwas schneller als das Verweiskonzept. Dies ist die Folge davon, dass beim Verweiskonzept zusätzlich zu den Datenrecords noch Verweisrecords gespeichert werden. Das Verfolgen von Verweisen bewirkt, dass Records häufiger aus Blöcken konvertiert werden müssen, was Rechenzeit kostet. Die Benutzung der Hauptspeicherabbildung ist demgegenüber etwas effizienter, da deren Verwaltung mit Hashing erfolgt.

Allerdings benötigt die Verwaltung der Abbildung viel Platz im Hauptspeicher, nämlich zwei Identifikatoren pro Record. Dies ist in der Praxis zu viel. Im Folgenden werden daher immer nur die Grafiken für das Verweiskonzept gezeigt, weil dieses Konzept bei guter Leistung deutlich weniger Hauptspeicher benötigt.

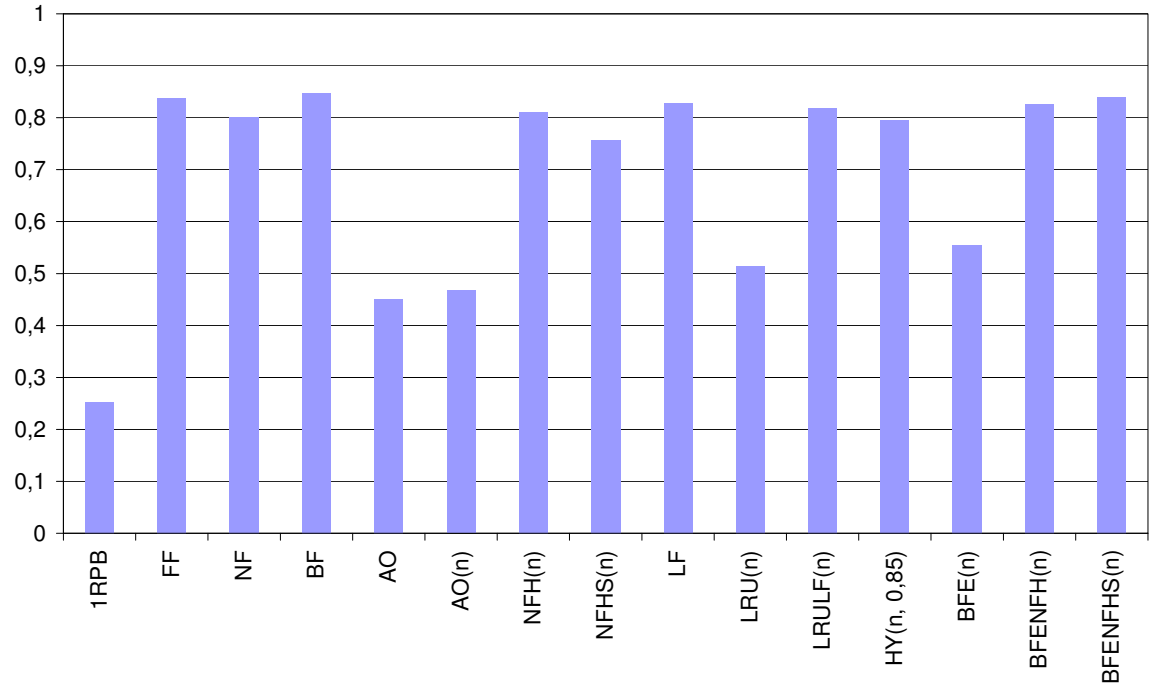


Abbildung 3.10: Speicherplatzausnutzung der verschiedenen Platzierungsstrategien. RM1, B=512, P=0, n=10

Speicherplatzausnutzung Die SPAN ist für alle Verfahren und die Szenarien RM1, RM2 und RM3 in den Abbildungen 3.10, 3.12 und 3.13 dargestellt. Bei RM1 sind die Verfahren 1RBP, AO und AO(n) erwartungsgemäß nicht brauchbar. Auch bei LRU(n) und BFE(n) reichen die gewählten Puffergrößen nicht aus, um auf annehmbaren SPAN-Werte zu kommen. Die quadratischen Verfahren zeigen immer eine gute SPAN. Knapp dahinter folgen die hybriden BFE-Verfahren und dann LRULF(n). Im Vergleich hierzu fällt die hybride AO-NFH Strategie deutlich um im Schnitt mehr als 5% nach unten ab.

BFE(n) schneidet bei RM1 wie gesagt schlecht ab, da $n = 10$ zu klein für die 500 hintereinander auftretenden Einfügeoperationen ist. Der Parameter n ist hier also kritisch. Dies zeigt Abbildung 3.11 noch einmal deutlicher. Die Variation des Parameters n hat in dem gezeigten Fall mit $B = 8192$ eine extrem große Auswirkung auf die SPAN. Bei $n = 5$ ist das Verfahren unbrauchbar ($SPAN < 30\%$), wohingegen bei $n = 20$ das Verfahren extrem gut ist ($SPAN > 90\%$).

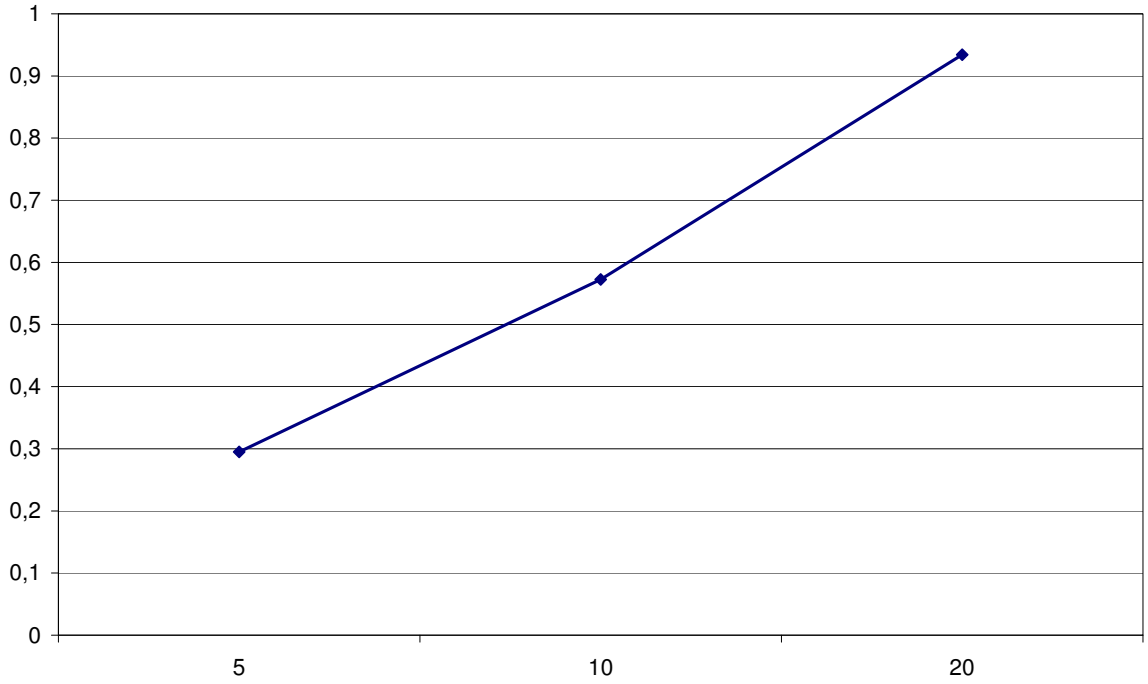


Abbildung 3.11: Speicherplatzausnutzung von BFE(n) für verschiedenes n. RM1, B=8192, P=0

Die Ergebnisse in Szenario RM2 (siehe Abbildung 3.12) unterscheiden sich ein wenig von RM1. Hier ist generell eine höhere SPAN von über 90% möglich. Auch hier ist 1RPC mit Abstand das schlechteste Verfahren. Alle anderen Verfahren liegen über 80%, wobei AO und HY(n,0.85) die Schlusslichter bilden. BFE(n) funktioniert bei dieser Anfragelast sehr gut. Bemerkenswert ist, dass NFH(n) etwa 2.7% besser abschneidet als das NFHS(n)-Verfahren, obwohl letzteres mehr Hauptspeicher benötigt.

Bei Szenario RM3 (Abbildung 3.13) zeigen sich die selben Tendenzen wie in Szenario RM1, allerdings liegen hier die Werte der SPAN deutlich niedriger. Diese Art der Last stellt also für einen Recordmanager ein größeres Problem dar. In diesem Szenario steigt bei dem Konzept der Identifikatoren mit Verweis die Anzahl der Verweise enorm an, wodurch eine optimale Speicherplatzausnutzung gar nicht erzielt werden kann.

Zugriffsverhalten auf dem Externspeicher Entscheidend für die Leistung des Recordmanagers sind i. d. R. Anzahl und Art von Externspeicher-

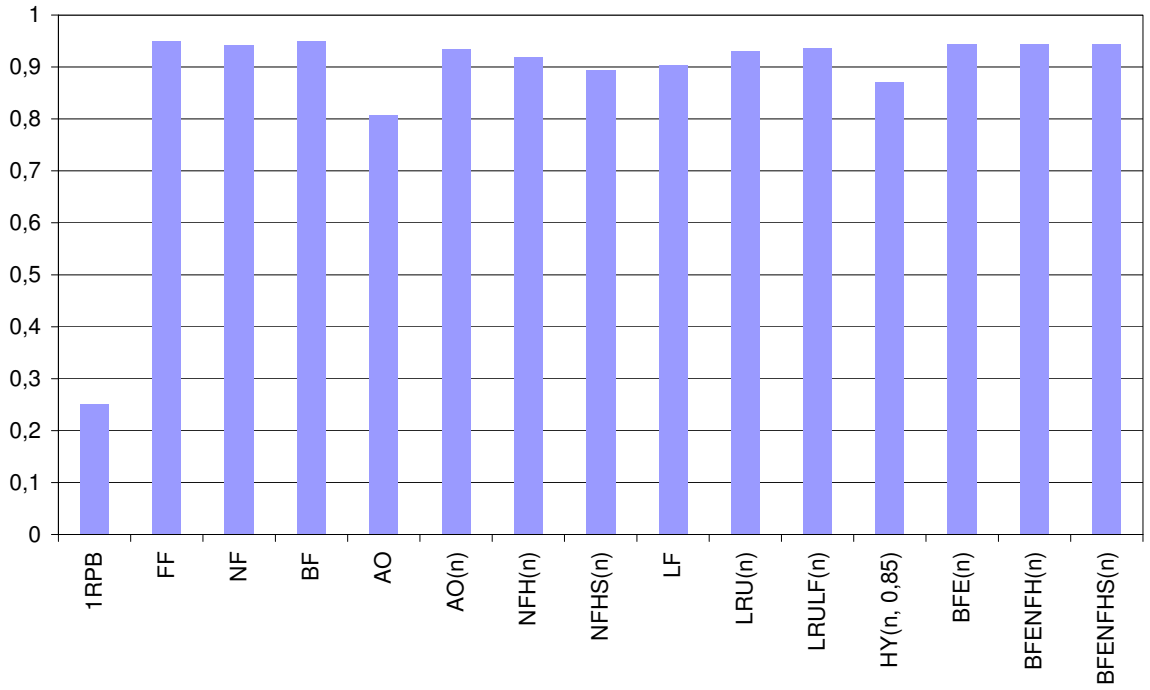


Abbildung 3.12: Speicherplatzausnutzung der verschiedenen Platzierungsstrategien. RM2, B=512, P=0, n=10

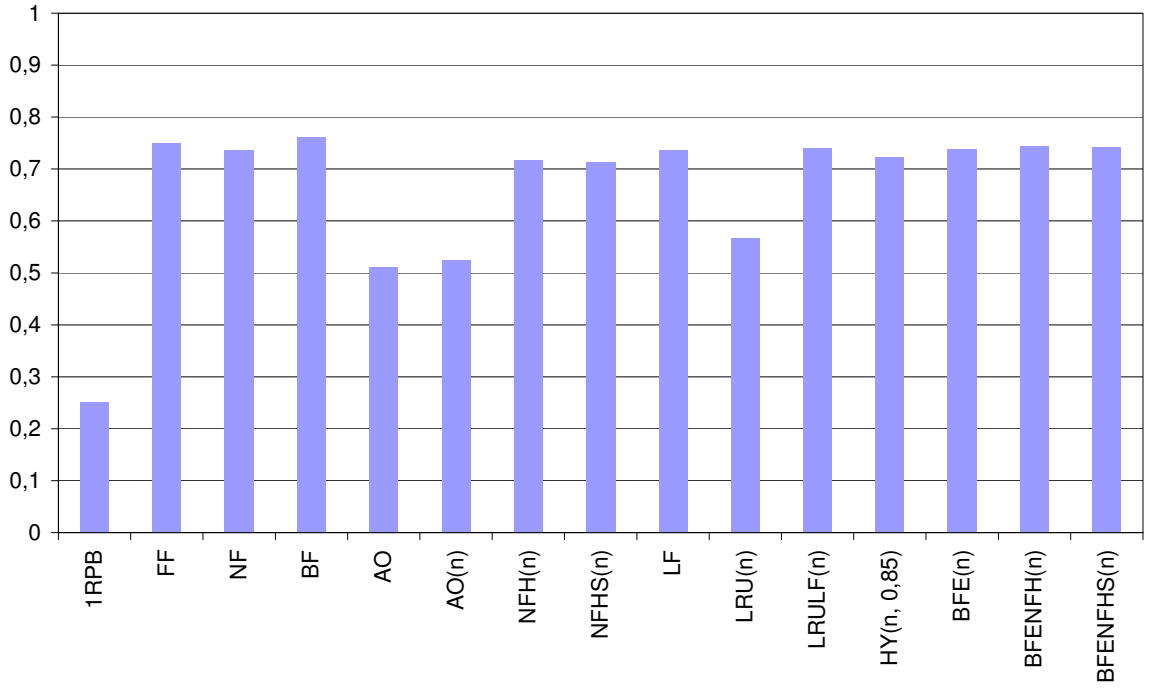


Abbildung 3.13: Speicherplatzausnutzung der verschiedenen Platzierungsstrategien. RM3, B=512, P=0, n=10

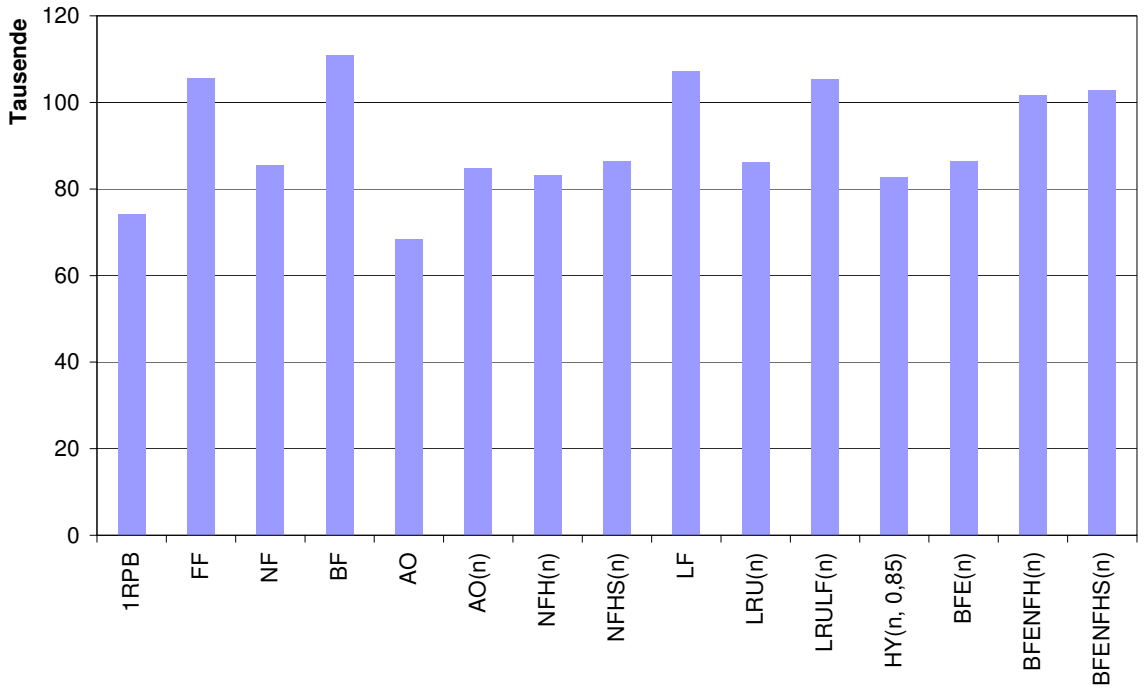


Abbildung 3.14: Anzahl der wahlfreien Externspeicherzugriffe. RM1, B=512, P=0, n=10

zugriffen. In Abbildung 3.14 sind die Anzahlen der wahlfreien Zugriffe (ohne Mehrfachzugriffe auf den gleichen Sektor) bei Szenario RM1 unter Benutzung der genannten Verfahren aufgetragen.

Es ist zu erkennen, dass AO und 1RBP hier am besten abschneiden. Der Grund dafür ist, dass in diesen Fällen neue Records niemals in bestehende Blöcke eingefügt werden. Es werden grundsätzlich neue Blöcke verwendet. Dass es hierbei überhaupt zu wahlfreien Zugriffen kommt, liegt daran, dass die Freispeicherverwaltung des Containers nicht nur Blöcke am bisherigen Ende des Containers zur Verfügung stellt, sondern auch Blöcke, die zwischenzeitlich wieder freigegeben wurden und daher irgendwo mitten im Container liegen können.

Alle jene Verfahren, die versuchen, neue Records in benachbarten Blöcken unterzubringen, schneiden bei den wahlfreien Zugriffen gut ab. Zu dieser Gruppe gehören NF, AO(n), NFH(n), NFHS(n) und HY(n,u). Auch BFE(n) kann man hierzu zählen, da die n betrachteten Blöcke in RM1 zu klein für das Einfügen von 1000 neuen Records war. FF führt ebenso zu vielen wahlfreien Zugriffen, da neue Records die Blöcke von vorne nach hinten auffüllen. Die Sprünge, die

der Plattenkopf ausführen muss, sind hierbei sicherlich nicht sehr groß, jedoch führt bereits die Rotationsverzögerung der Festplatte zu längeren Wartezeiten.

Alle BF-Verfahren erzeugen eine große Menge an wahlfreien Zugriffen. Dies liegt daran, dass es selten ist, dass zwei bei den Tests nacheinander zugegriffene Records zuvor in benachbarten Seiten untergebracht wurden.

In diesem Experiment lag bei allen Verfahren die Anzahl der sequentiellen Zugriffe deutlich unter der Anzahl der wahlfreien Zugriffe. Berücksichtigt man den Faktor von mindestens zehn zwischen wahlfreien und sequentiellen Zugriffen, so ist der Einfluss der sequentiellen Zugriffe verschwindend gering. Für die Beurteilung der Verfahren spielen sie keine Rolle. Auf eine Grafik wurde daher verzichtet.

Wenn ohne Puffer gearbeitet wird, steigt jedoch die Anzahl der Zugriffe, die unmittelbar nacheinander den selben Sektor anfragen (Mehrfachzugriffe). Dies liegt daran, dass beim Recordmanager häufig erst ein Block gelesen werden muss, dann Änderungen durchgeführt werden und anschließend das Zurückschreiben erfolgt. Solche Zugriffe auf den selben Sektor sind noch schlimmer zu bewerten als Seeks auf benachbarte Zylinder, denn hier in jedem Fall eine volle Umdrehung der Platte abgewartet werden. Die Messwerte der Zugriffe auf den selben Sektor sind in Abbildung 3.15 zu finden. 1RBP baut hierdurch seinen Vorsprung in der Disziplin Externspeicherzugriff weiter aus.

In der Summe zeigt sich, dass HY(n,0.85) etwa 8.5 Prozent besser als die hybriden BF-Strategien abschneidet, während LRULF(n) noch knapp dahinter rangiert. Das bisher vorgestellte Szenario berücksichtigt jedoch nicht den Einfluss von Puffern. Jedes Datenbanksystem muss Blöcke im Hauptspeicher zwischenpuffern, um die gewünschten Operationen effizient zu unterstützen. Normale Puffergrößen liegen zwischen einem und 20 Prozent der Größe der Datenmengen. Bei Anfragelasten mit viel sequentiellen E/A-Operationen sind kleinere Puffer ausreichend, während bei Anfragelasten wie RM1 mit viel wahlfreien Zugriffen größere Puffer nötig sind.

Das eben vorgestellte Experiment wurde nun mit einem LRU-Puffer der Größe 10% wiederholt (siehe Abbildung 3.16). Es zeigen sich hierbei einige interes-

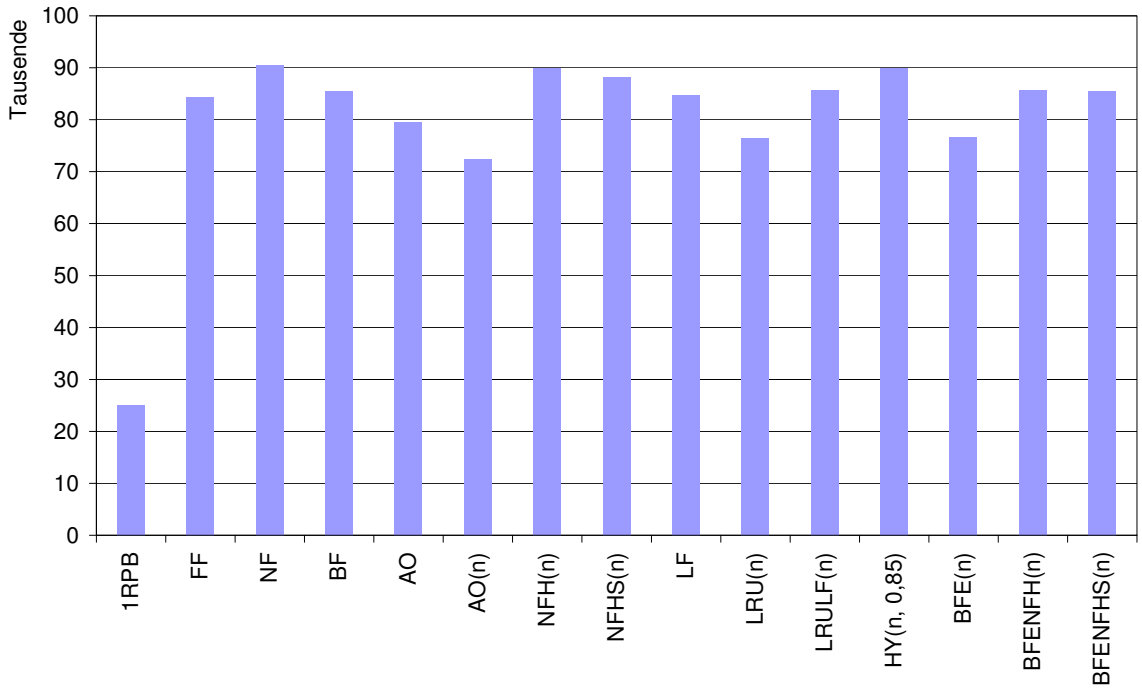


Abbildung 3.15: Anzahl der Mehrfachzugriffe von Blöcken auf dem Externspeicher. RM1, B=512, P=0, n=10

sante Effekte, die ohne Puffer nicht auftreten. Zum einen verschwinden die wiederholten Zugriffe auf den selben Sektor nahezu vollständig. Aus diesem Grund erhöhen sich allerdings die wahlfreien Zugriffe erheblich.

Verfahren, die insgesamt weniger Blöcke benutzen, also eine höhere SPAN aufweisen, können Blockpuffer deutlich besser nutzen, als Verfahren mit niedriger SPAN. Dies liegt daran, dass die Wahrscheinlichkeit für einen Puffertreffer bei weniger Blöcken deutlich höher liegt. Aufgrund dieser Tatsache schieben sich die hybriden BF-Verfahren und LRULF(n) an HY(n,u) im benutzten Szenario mit RM1 vorbei. Unter Berücksichtigung aller Messungen kann beim Zugriffsverhalten jedoch kein großer Unterschied zwischen diesen Verfahren gefunden werden. Bei einigen Messungen liegt das eine Verfahren vorne, bei anderen das andere.

Szenario RM4 Eine Strategie für die Platzierung von Records in einem Recordmanager muss sich in vielen Disziplinen beweisen. Je nach Anwendung kann ein schlechter Wert in einer einzigen Disziplin bereits das K.O.-Kriterium

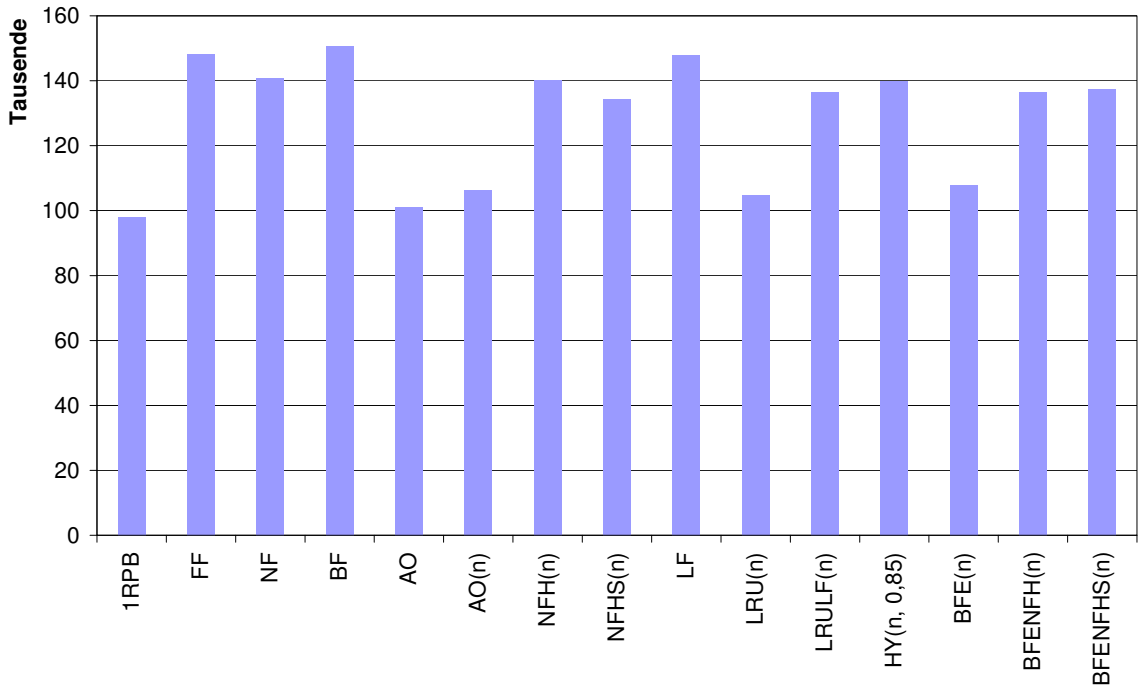


Abbildung 3.16: Anzahl der wahlfreien Externspeicherzugriffe unter Berücksichtigung einer Pufferung. RM1, B=512, P=10%, n=10

sein. An dieser Stelle wird bereits eine erste Auswahl an Strategien getroffen, da die Darstellung von allen Messergebnissen nicht unbedingt neue Erkenntnisse bringt. Aufgrund der oben beschriebenen Resultate sollen bei Szenario RM4 nur die Platzierungsstrategien AO, BFE(10), HY(10,0.85), LRULF(10), BFENFH(10) und BFENFHS(10) betrachtet werden. Die Messungen wurden selbstverständlich für alle Strategien durchgeführt, jedoch enthalten die Messergebnisse für die anderen Strategien keine weitere, interessante Aussagen.

Bei RM4 werden Einfüge- und Änderungsoperationen gemischt durchgeführt. In den Experimenten wird das Verhältnis zwischen Einfüge- und Änderungsoperationen über den Parameter p_a gesteuert. Bei $p_a = 0$ werden keine Änderungsoperationen ausgeführt, währenddessen bei $p_a = 100$ ausschließlich Änderungsoperationen erfolgen. Änderungsoperationen sind i. d. R. aufwendiger als Einfügeoperationen, weshalb die Zugriffskosten mit größer werdendem p_a ansteigen.

In Abbildung 3.17 ist die Speicherplatzausnutzung der verschiedenen Strategien im Verhältnis zur Änderungsrate p_a aufgetragen. Zu erkennen ist, dass die

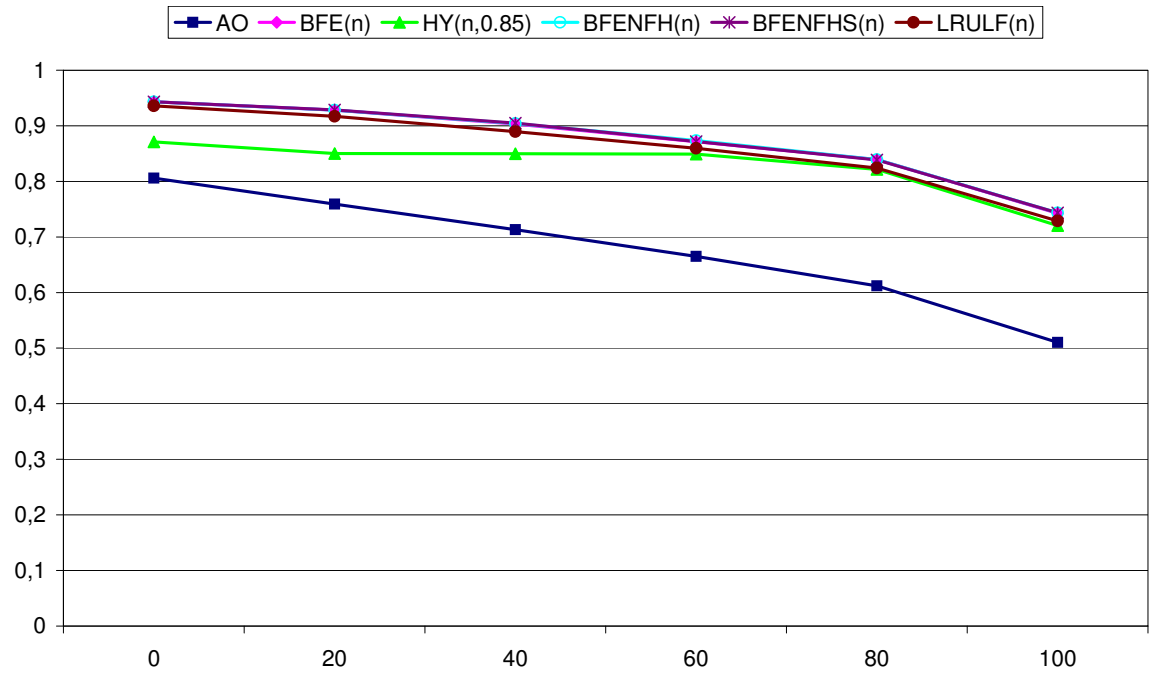


Abbildung 3.17: Speicherplatzausnutzung bei verschiedenen Änderungsraten p_a. RM4, B=512, P=10%, n=10

SPAN mit steigendem p_a merklich sinkt, bei den besten Verfahren von 95% auf 75%. AO liefert wie erwartet die schlechtesten Werte. Für Änderungsraten kleiner als 60% schneidet HY(10,0.85) schlechter als die anderen Strategien ab, da die vorgegebenen 85% SPAN nicht deutlich übertroffen werden können. Die anderen Strategien liegen eng beisammen. LRULF(10) ist minimal schlechter als die BFE-Verfahren.

Die eigentlichen Kosten für RM4 entstehen durch wahlfreie Zugriffe auf den Externspeicher. Zugriffe auf den selben Sektor werden durch den verwendeten Puffer nahezu auf null reduziert und sequentielle Zugriffe spielen wie bereits in den Experimenten mit RM1 bis RM3 kaum eine Rolle. Die CPU-Kosten sind für LRULF(10) leicht höher, als für die anderen Verfahren, jedoch können diese gut vernachlässigt werden.

Abbildung 3.18 zeigt die Anzahlen der wahlfreien Externspeicherzugriffe, die durch RM4 entstehen. Insgesamt werden bei RM4 1000 Einfügeoperationen und 25000 weitere, eventuell gemischte Einfüge- und Änderungsoperationen durchgeführt. Wie zu erwarten war, steigen die Kosten mit wachsendem p_a

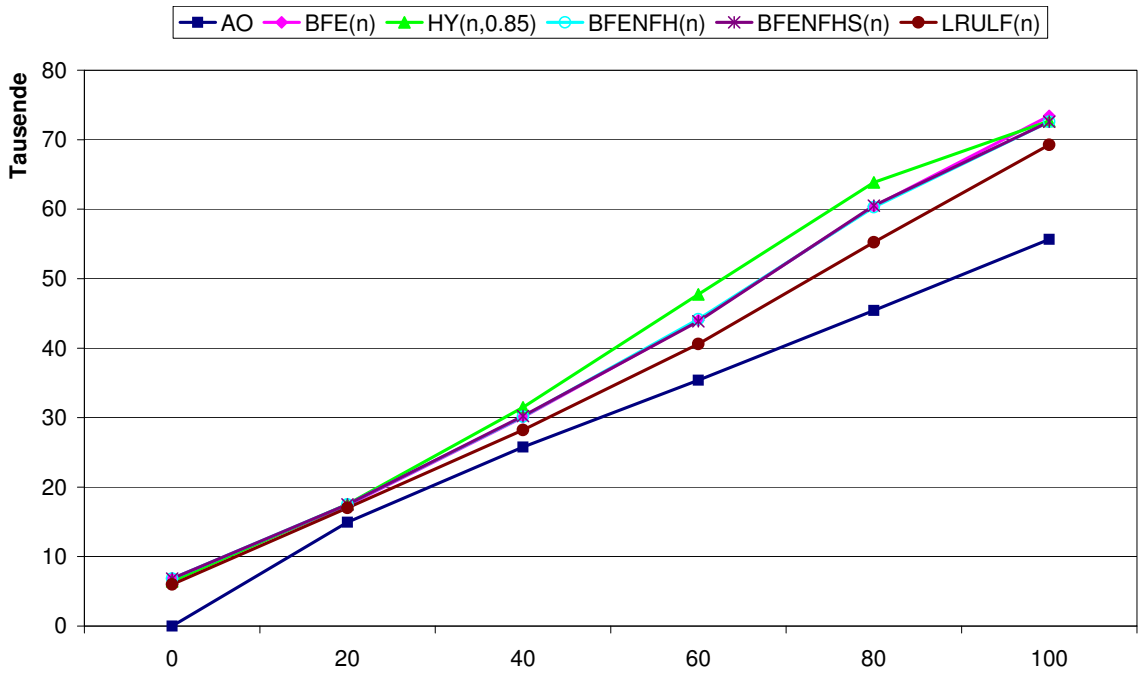


Abbildung 3.18: Anzahl der wahlfreien Externspeicherzugriffe bei verschiedenen Änderungsraten p_a. RM4, B=512, P=10%, n=10

deutlich an. AO liegt in dieser Disziplin vorne, da die Einfügeoperationen sehr viele sequentielle Zugriffe produzieren. HY(10,0.85) zeigt die schlechtesten Eigenschaften bezüglich wahlfreier Zugriffe, da sehr oft die NF-Teilstrategie zur Erhaltung der Speicherplatzausnutzung arbeiten muss.

Um einige Prozente besser sind die BFE-Strategien. Ihre Kurven liegen praktisch aufeinander. Für die Anfragelast ist $n = 10$ groß genug gewählt, wodurch die Nextfit-Komponenten nur selten Verwendung finden. Somit wird bei beiden Strategien nur der gemeinsame BFE-Anteil genutzt. Der eigentliche Sieger in Szenario RM4 ist jedoch LRULF(n), das bei höheren Änderungsraten um bis zu 5% bessere Werte liefert als BFE.

Bei größeren Blockgrößen schneidet HY(10,0.85) in Szenario RM4 besser ab als in den dargestellten Experimenten mit $B = 512$. Dies liegt daran, dass das zugrunde liegende AO(n) bei im Vergleich zur Blockgröße kleineren Records eine bessere Speicherplatzausnutzung garantieren kann. Hierdurch wird NFH(n) sehr viel seltener innerhalb der hybriden Strategie verwendet, da AO(n) allein bereits die anvisierten 85% SPAN erreicht. Unsere realen Szenarien sind

jedoch näher an dem dargestellten Fall mit $B = 512$, in dem die Größe der Records im Schnitt bei einem Viertel der Blockgröße liegt.

In den folgenden Kapiteln wird an einigen Stellen die Auswirkung der Platzierungsstrategie des Recordmanagers bei Experimenten mit Realdaten untersucht. Um die Anzahl der durchzuführenden Experimente einzuschränken und trotzdem verschiedene Effekte beobachten zu können wurden die folgenden Strategien für diese weiteren Tests ausgewählt: AO, HY(10,0.85), LRULF(10) und BFENFH(10).

Fazit Es gibt momentan nicht die eine Platzierungsstrategie, die in allen Fällen immer das beste Verhalten zeigt. Dies liegt daran, dass die Definition von „bestem Verhalten“ von Applikation zu Applikation sehr unterschiedlich ist. Die eine Applikation erfordert eine hohe Speicherplatzausnutzung, die andere extrem wenige wahlfreie Festplattenzugriffe. Generell ist eine hohe Speicherplatzausnutzung allerdings nicht ohne wahlfreie Festplattenzugriffe zu erreichen, weshalb sich diese Ziele teilweise widersprechen.

Dennoch können einige generelle Aussagen gemacht werden. Es hat sich gezeigt, dass die aus der Literatur bekannte hybride Strategie HY(n,u) [MCS96] zwar in vielen Bereichen gute Werte liefert, jedoch kommt sie an die neuen hybriden Strategien mit BFE-Technik nicht heran. Besonders bei der Speicherplatzausnutzung ist der Vorteil der BFE-Verfahren groß. Weiterhin stellt die Festlegung des Parameters u bei HY(n,u) ein Risiko dar, das bei den BFE-Strategien nicht eingegangen werden muss.

Die LRULF(n)-Strategie kann ebenfalls für viele Zwecke empfohlen werden. Jedoch hat sie deutliche Schwachstellen in der Rechenlast bei Szenario RM2 gezeigt und ist daher für Lasten mit vielen Einfügeoperationen in schneller Abfolge nur bedingt zu gebrauchen.

Interessant wäre noch eine Erweiterung der BF-Strategie. Verwendet man einen dem freien Speicherplatz sortierten binären Suchbaum, so könnte der CPU-Aufwand bei der Suche deutlich reduziert werden bei ansonsten gleichen Eigenschaften des Verfahrens. Allerdings ist so ein Suchbaum im Vergleich zu den n-elementigen Strukturen der anderen Strategien sehr groß. Daher wurde auf die Implementierung dieser Erweiterung verzichtet.

Besondere Erwähnung verdient ein Merkmal des Recordmanagers von XXL. Er ist dazu in der Lage, die Strategien zur Laufzeit auszutauschen. Allerdings muss hierbei eine Warmlaufphase der neuen Strategie einkalkuliert werden. Der Recordmanager kann beispielsweise für kurze Zeit auf AO eingestellt werden, um für einen Algorithmus, der ausschließlich Records einfügt, die optimale Leistung zu liefern. Anschließend wird wieder auf eine Strategie zurückgeschaltet, die bei normaler Last eine gute SPAN und niedrige Zugriffszahlen garantiert.

3.2.4 Das Datenbank-Framework

Für einige weitere Tests in den folgenden Kapiteln ist es sinnvoll, eine kleine Datenbankumgebung auf Basis der Funktionalität von XXL aufzubauen. In XXL existiert bereits eine ganze Menge von Funktionalität, die für die Entwicklung eines Datenbanksystems nur noch miteinander gekoppelt werden muss. Das im Rahmen dieser Arbeit entwickelte, so genannte Datenbank-Framework erlaubt es, Datensätze in Records zu speichern, diese über verschiedene Indexstrukturen zu indexieren und Anfragen zu formulieren.

Bislang wird das Datenbankframework über Java-Methodenaufrufe gesteuert. Anfragen können unter Benutzung der XXL GUI in ihrer Operatorbaumdarstellung graphisch eingegeben werden. Hieraus wird dann Java-Quelltext erzeugt, der in eigenen Applikationen eingesetzt werden kann. Eine Nutzung von Datenbankstandards, d. h. eine Unterstützung von Teilen von SQL und/oder JDBC soll in Zukunft ermöglicht werden.

Der Sinn des Datenbank-Frameworks ist der, dass nun Verfahren aus der Literatur besser miteinander verglichen werden können. Viele abstrakt formulierte Algorithmen benutzen Datenbanksysteme als Grundlage. Diese konnten bislang nicht mit Verfahren verglichen werden, die auf dem Dateisystem oder mit direktem Festplattenzugriff arbeiten. Bislang waren nur solche Systemvergleiche möglich, bei denen die Laufzeit des compilierten Binärcodes gemessen wurde. Hierbei haben jedoch beispielsweise Laufzeitumgebungen, Compiler und Implementierungssprachen große Auswirkungen auf das Messergebnis. Dies ist natürlich nicht gewünscht, weil eigentlich Aussagen über die Effizienz der eingesetzten Verfahren gemacht werden sollen.

XXL bietet jetzt die Möglichkeit, beide Arten von Verfahren auf der Basis einer umfassenden Werkzeugunterstützung zu implementieren. Anschließend können die Verfahren anhand von Kriterien wie Externspeicherzugriff und Laufzeit fair in der selben Laufzeitumgebung miteinander verglichen werden.

Das Datenbank-Framework wurde also nicht entwickelt, um den bereits vorhandenen open-source Datenbanksystemen ein weiteres System hinzuzufügen, sondern ganz allein, um das Testframework von XXL zu erweitern. In Zukunft könnte das System jedoch in die eine oder andere forschungsrelevante Richtung weiterentwickelt werden, wodurch ein wirklich neuartiges System entstehen würde.

3.2.5 Die XXL Architektur

Die Basiskomponenten von XXL, die der Speicherung von Daten dienen, sind so konzipiert, dass sie sich verschiedenen logischen Schichten zuordnen lassen. Abbildung 3.19 enthält eine Übersicht über diese Art von Funktionalität in XXL. In den folgenden Kapiteln werden weitere Schichten hinzugefügt, die für eine XML Datenbank notwendig sind.

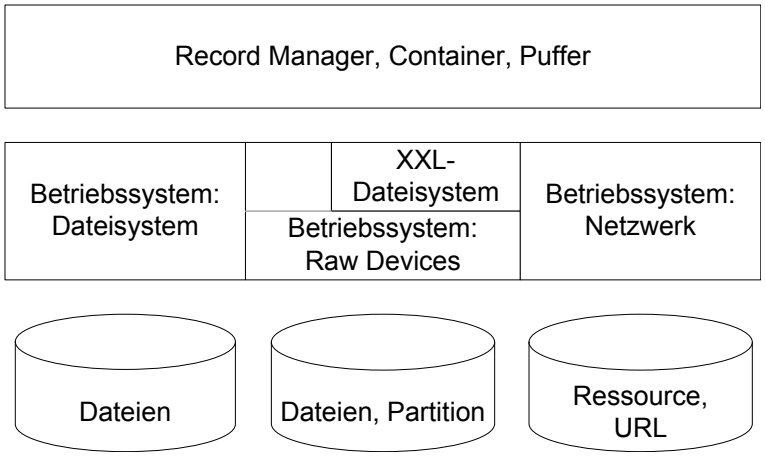


Abbildung 3.19: Die Ebenenarchitektur in der Speicherungsschicht von XXL

In einer realen Anwendung treten normaler Weise eine ganze Reihe von Objekten auf, die eine entsprechende Funktionalität für die Speicherung anbieten.

Ein Beispiel einer Speicherungshierarchie zeigt Abbildung 3.20. Hier sind auch die wichtigsten Parametrisierungen der Objekte in den Ovalen zu finden.

In dem Beispiel greift eine Anwendung auf Objekte beliebiger Länge zu. Dabei kommuniziert sie nur mit dem oberen `BufferedContainer` direkt (mit dem so genannten Objektpuffer). Dieser leitet die Anfragen dann bei Bedarf an die darunter liegenden Objekte weiter. In der Schicht darunter wird das Objekt also mittels eines Converters in eine Byterepräsentation konvertiert bzw. das Objekt aus einer Byterepräsentation erzeugt. Die Byterepräsentation wird in der nächsten Schicht dem Recordmanager übergeben. Der Recordmanager führt bei Bedarf Blockzugriffe auf dem Blockpuffer durch, der seine Daten wiederum in einem RawAccessContainer verwaltet. Der RawAccessContainer nutzt den direkten Festplattenzugriff über die RawAccess-Schnittstelle.

Es mag auf den ersten Blick erschrecken, wie viele Ebenen hier benutzt werden, jedoch ist die Anzahl der Ebenen nicht kritisch. Die Ebenen unterhalb des RawAccessContainers sind sowieso immer hinter normalen Dateisystemoperationen des Betriebssystems verborgen. Die Pufferungsebenen sorgen dafür, dass Anfragen möglichst weit oben und möglichst ohne wahlfreie Zugriffe, d. h. performant beantwortet werden können.

3.3 Zusammenfassung und Ausblick

In diesem Kapitel wurden wichtige Basiskomponenten beschrieben, die in der Java-Bibliothek XXL vorliegen und auf die in den nächsten Kapiteln zurückgegriffen wird. Aufgrund der Erfordernisse von Datenbanksystemen muss XXL eine Möglichkeit bieten, die vollständige Kontrolle über die Zugriffe auf die Festplatte zu erhalten. Die Natur der Programmiersprache Java führt dazu, dass die Kontrolle nur über Umwege zurückerhalten werden kann. In diesem Kapitel wurden die entsprechenden Komponenten erläutert und anschließend evaluiert.

Der weitere Schwerpunkt dieses Kapitels lag auf dem Recordmanager von XXL, in dem viele Platzierungsstrategien aus der Literatur implementiert wurden. Darüber hinaus wurden neue Strategien entwickelt. Diese besitzen deutliche

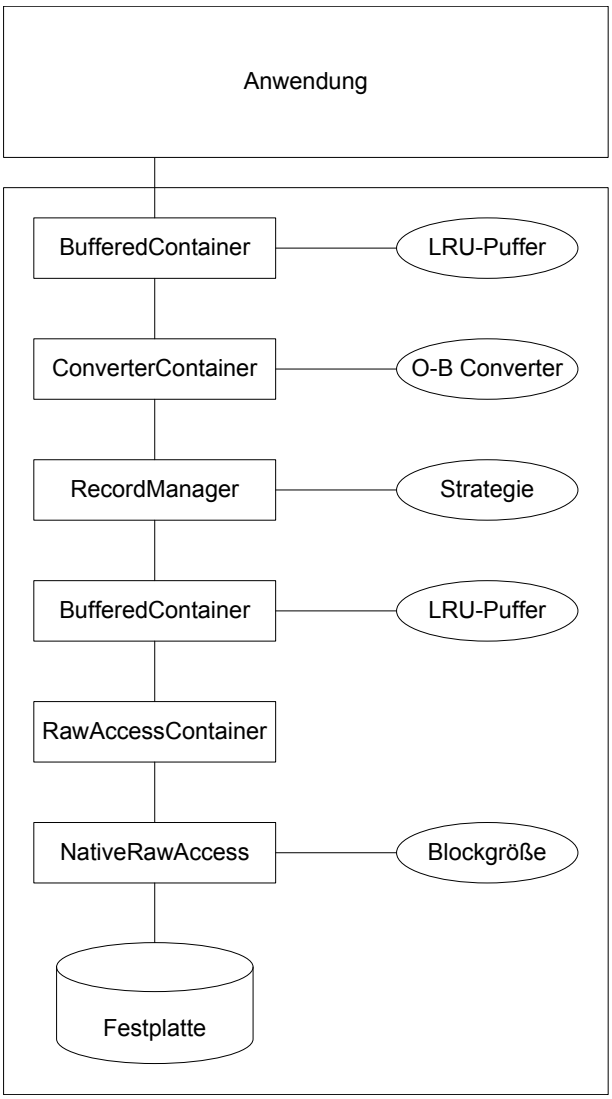


Abbildung 3.20: Beispiel für eine Hierarchie von Objekten zur Speicherung von Records

Vorteile in einigen Kriterien. Jedoch gilt, dass für verschiedene Arbeitslasten auch verschiedene Strategien die besten Ergebnisse liefern. Hierzu bietet der Recordmanager von XXL die Möglichkeit des Austauschs zur Laufzeit.

Im nun folgenden vierten Kapitel werden einige der Basiskomponenten von XXL dazu verwendet, um datenzentrisches XML in Relationen umzuwandeln. Im fünften Kapitel werden dann noch sehr viel mehr der vorgestellten Komponenten für den Aufbau eines nativen XML Speichers eingesetzt. Dieser Speicher ist besonders für dokumentzentrisches XML geeignet.

Kapitel 4

XML und relationale Datenbanken

Neben der Speicherung von semi-strukturierten Daten hat sich XML, wie bereits im ersten Kapitel erwähnt, als Standard für den Austausch von voll strukturierten Daten zwischen relationalen Systemen etabliert. Datenbanken werden zunächst in eine XML Repräsentation überführt, diese als Datei transferiert und auf einem Zielrechner wieder in das relationale Modell gebracht.

Bei der Umsetzung von relationalen Datenbanken nach XML gibt es sehr viele Möglichkeiten [TDCZ02]. Die Internetseite von Ronald Bourret [Bou04] ist eine der bekanntesten und vollständigsten Quellen, die einen Überblick über existierende Verfahren geben. Dort werden auch eine ganze Reihe von freien und kommerziellen Hilfsprogrammen beschrieben und miteinander verglichen.

Häufig ist die Technik von so genannten *Abbildungsdateien* anzutreffen. In einer Datei wird eine Abbildung definiert, auf deren Basis die Hin- und Rücktransformation durchgeführt wird. Die Abbildung kann bei Kenntnis des Datenbankschemas automatisch generiert werden. Hierbei werden aus dem Datenbankschema sowohl eine spezielle XML Grammatik (DTD oder XSD), als auch eine Konfigurationsdatei für die Abbildung erzeugt.

Eine davon abweichende Möglichkeit bieten XML Sprachen, die allgemein jede Datenbank beschreiben können. Hier wird keine Abbildung definiert, und die Grammatik für die XML Daten steht von Anfang an fest.

Abbildungen zwischen XML und relationalen Datenbanken wurden schon in vielen Projekten betrachtet. Inzwischen bleiben nur wenige Wünsche in der Praxis offen. Allerdings stellen sich in diesem Zusammenhang weitere Fragen in Bezug auf die Leistung der Verfahren und deren Anwendbarkeit. Im Folgenden soll darauf näher eingegangen werden.

4.1 Relationales XML als Datenquelle für XXL

In XXL sollte die direkte Integration von XML Datenquellen in die relationale Anfrageverarbeitung ermöglicht werden. Hierzu sollten Eingabe-Cursor entwickelt werden, die eine möglichst allgemein gültige Abbildung von XML in relationale Tupel erlauben und trotzdem noch effizient arbeiten.

Das Grundprinzip der Cursor-Algebra von XXL (siehe Kapitel 3.1.2) ist das der bedarfsgesteuerten Auswertung. Eine Operation wird erst dann ausgeführt, wenn von außen das nächste Element des Ergebnisses gewünscht wird und dies die Berechnung der jeweiligen Operation notwendig macht. Die für XML gewünschten Cursor sollten auf jeden Fall dieses Prinzip der Bedarfssteuerung verwenden. Im XML Bereich wurden solche Operatoren in der Literatur bislang noch nicht beschrieben. Jedoch hat die Bedarfssteuerung erhebliche Auswirkungen auf Aufbau und Leistung der Verfahren, was später in diesem Kapitel verdeutlicht wird.

Die in XXL im Paket `xxl.core.xml.relational` implementierten Operatoren unterscheiden zwischen Daten, die die Attributwerte der eigentlichen Tupel enthalten, und Metadaten, wie beispielsweise Attributnamen und Datentypen. Zu ihrer Lokalisation wird für Daten und Metadaten jeweils ein XPath Ausdruck angegeben. Fehlt der XPath-Ausdruck für Metadaten, so werden automatisch Standardmetadaten aus dem ersten gefundenen Tupel generiert.

Weiterhin kann angegeben werden, welche XML Knoten ein Tupel bzw. einen Attributwert kennzeichnen. Auch die Attributnamen, unter denen man gewisse Metadaten finden kann, sind flexibel einstellbar. Ein Beispiel für eine unterstützte XML Datei ist in Abbildung 4.1 zu finden.

```
<?xml version="1.0" encoding="UTF-8"?>
<databases>
   <web>
      <meta>
         <col sqltype="LONGCHAR">Webseite</col>
         <col sqltype="VARCHAR">Beschreibung</col>
         <col sqltype="NUMERIC" precission="10">GroesseKB</col>
      </meta>
      <data>
         <row>
            <col>http://dbs.mathematik.uni-marburg.de</col>
            <col>Webseite der Datenbankgruppe</col>
            <col>10000</col>
         </row>
         <row>
            <col>http://www.maschn.de</col>
            <col>Private Webseite von Martin Schneider</col>
            <col>200</col>
         </row>
      </data>
   </web>
</databases>
```

Abbildung 4.1: Beispiel für relationales XML

Eine sehr beliebte Anwendung dieser Operatoren ist das Lesen von relationalen Daten aus Tabellen innerhalb von HTML-Dateien aus dem Internet. Solche Daten sind an sehr vielen Stellen im Internet vorhanden, seien es beispielsweise Listen von Angeboten bei einem Internetshop, Adressen- und Telefonlisten von Firmen und Organisationen oder die Ergebnislisten von Sportveranstaltungen. Es existiert eine sehr große Anzahl solcher Quellen, auf deren Basis der Anwender gerne mit zielgerichteten SQL-Anfragen Informationen gewinnen will.

Moderne Datenbanksysteme müssen das Internet als Datenquelle direkt nutzbar machen, da nur so aktuelle Daten in Anfragen akkurat benutzt werden können. Gleichzeitig ist die verfügbare Bandbreite des Kommunikationskanals zu solchen Datenquellen unter Umständen relativ niedrig, weshalb eine bedarfs-

gesteuerte Auswertung sinnvoll ist. Außerdem kann es dem Anwender wichtig sein, wenn er erste Ergebnisse relativ früh erhält. Eventuell interessiert sich der Anwender sogar überhaupt nicht für die vollständige Berechnung aller Ergebnisse.

In XXL wurden zwei verschiedene XML Einleseoperatoren implementiert, die nach unterschiedlichen Prinzipien arbeiten. In Kapitel 2.4.1 wurde bereits darauf eingegangen, dass zwei vollständig verschiedene Standards für das Parsen von XML Dokumenten existieren. Auf Basis dieser Parser wurde nun jeweils ein Operator entwickelt. Eine ausführliche Beschreibung der beiden Operatoren ist in den folgenden Abschnitten zu finden.

4.1.1 DOM

Bei der DOM-Variante wird zunächst das komplette XML Dokument geparst und dabei der XML Baum im Hauptspeicher aufgebaut. Dabei entstehen viele kleine Objekte, wodurch dieser Schritt relativ aufwendig ist. Als erstes werden die Metadaten benötigt. Diese werden mittels einer erhältlichen XPath-Implementierung (hier: Apache Xalan [Fou04b]) im Dokument gesucht und anschließend in eine entsprechende Form gebracht (`ResultSetMetaData`).

Anschließend wird mit der selben Methode der erste Datensatz gesucht und ein Positionsmarker auf ihn intern gespeichert. Erst, wenn eine Anforderung für ein Tupel von außen kommt, wird ein Tupel erzeugt und zurückgegeben. Danach muss selbstverständlich der Positionsmarker weitergesetzt werden, wofür jetzt jedoch lokale Navigationsoperationen der DOM-API ausreichend und effizient sind. Es muss also nicht jedes Mal neu von der Wurzel ausgehend mit XPath gesucht werden.

4.1.2 SAX

Das SAX-Parser Modell unterscheidet sich von DOM besonders im Punkt der Hauptspeichernutzung. Es ist zu keinem Zeitpunkt das komplette Dokument im Hauptspeicher vorhanden. Im Gegenteil: Wenn nicht gleichzeitig noch eine

Validierung vorgenommen wird, reicht Hauptspeicher in der Größenordnung des größten Knotens. Das XML Dokument wird sequentiell gelesen und bei Erkennung eines Knotens wird ein Ereignis generiert. Dieses Ereignis wird per Methodenaufruf an eine Verarbeitungsschnittstelle, den so genannten *SAX Eventhandler*, weitergeleitet.

Ein Entwurfsziel dieses Einleseoperators war entsprechend die Implementierung eines Verfahrens, das mit möglichst wenig Hauptspeicher auskommt. Dieses Entwurfsziel war beim DOM-Operator nicht vorgegeben, da der Parser selbst bereits Hauptspeicher in der Größenordnung des Eingabedokuments belegt.

Das erste zu lösende Problem war die Erkennung der Lokation durch einfache XPath-Ausdrücke. Bei SAX-Parsern existiert keine standardisierte Anbindung einer XPath-Anfrageauswertung. Zu diesem Zweck wurde ein spezieller, dekorierender SAX Eventhandler geschrieben. Dieser sucht im SAX Ereignisstrom nach einfachen XPath-Ausdrücken. An dieser Stelle unterstützen wir XPath-Ausdrücke, die auf die descendant und descendant-or-self Achsen beschränkt sind und nur Prädikate bestehend aus einer Knotennummer enthalten.

Wird ein Knoten k gefunden, der dem gesuchten XPath Ausdruck entspricht, so werden von nun an die SAX Ereignisse an einen zweiten Eventhandler weitergeleitet. Das Senden der Ereignisse wird erst wieder eingestellt, sobald die zu k gehörende Endmarkierung gefunden wird. Somit erhält der zweite Eventhandler exakt diejenigen Ereignisse, die zu dem Teilbaum gehören, dessen Wurzel der gefundene Knoten k ist.

Zunächst müssen auch bei der SAX-Variante die Metadaten extrahiert werden. Dazu wird der Parser gestartet, und bei Erreichen der Metadaten erzeugt ein spezieller Metadaten Eventhandler die gewünschten Metadaten. Problematisch ist dies, falls die Metadaten im Dokument hinter den Daten liegen (in Dokumentenreihenfolge). In diesem Fall wurden die Daten zum Zeitpunkt der Extraktion der Metadaten bereits übersprungen und das XML Dokument muss zur eigentlichen Datenextraktion ein zweites Mal gelesen werden. Dadurch wird die Leistung des Verfahrens deutlich negativ beeinflusst. Dieser Fall, dass die Metadaten hinter den Daten stehen, ist allerdings sehr selten, da relationale XML Daten normaler Weise dazu erzeugt werden, um sie anschließend problemlos wieder in relationale Systeme importieren zu können.

Ein weiteres Problem ergibt sich durch die Vorgabe der Bedarfssteuerung. Der SAX-Parser liefert Ereignisse, auf die der Eventhandler reagieren kann. Von der Seite des Operatorbaums kommen Anfragen nach fertigen Tupeln (ONC-Prinzip), was ebenfalls als Senden von Ereignissen verstanden werden kann. Die Zwischenschicht, muss also zwischen den verschiedenen, zunächst nicht synchronisierten Ereignissen vermitteln.

Die implementierte Lösung dieses Problems zeigt schematisch Abbildung 4.2. Der SAX Einleseoperator startet den SAX-Parser in einem eigenen Thread. Der SAX-Parser erzeugt seine Ereignisse und sendet diese über den XPath Dekorator Eventhandler an den Tupelextraktor Eventhandler. Dieser sendet seine Tupel über einen unidirektionalen, asynchronen Kanal an den Einleseoperator zurück. Es sei noch bemerkt, dass der XPath Dekorator auf Weiterleitung geschaltet werden kann, sobald die eigentlichen Daten gefunden wurden. Dies ist deutlich effizienter, weil der Tupelextraktor dann die eigentlichen Tupel ohne komplette XPath Auswertung erzeugen kann (analog zum DOM Operator).

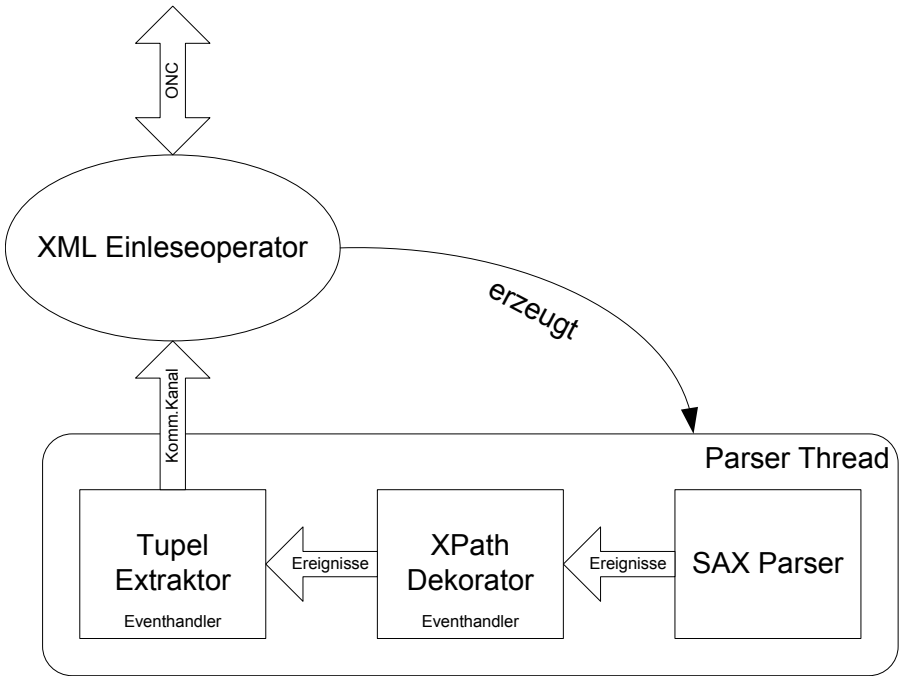


Abbildung 4.2: Der Aufbau des SAX Einleseoperators

Der Einleseoperator kann somit den Parser über die Benutzung des Kommunikationskanals steuern. Wenn der Kanal voll ist, so wird der Tupelextraktor und

damit der komplette Parserthread blockiert. Der Parserthread kann erst wieder weiterlaufen, wenn Tupel vom Einleseoperator entnommen wurden. Versucht anders herum der Einleseoperator die Tupel schneller zu lesen, als sie vom Tupelextraktor geliefert werden, so wird er und somit der komplette Operatorbaum blockiert.

4.1.3 Evaluierung

In diesem Abschnitt erfolgt eine Evaluierung der beschriebenen XML Einleseoperatoren anhand von Realdaten. Messbare Kriterien für die Bewertung der Operatoren sind die Tupelleistung (gemessen in Tupel pro Sekunde oder KB pro Sekunde), sowie Hauptspeicherverbrauch und Zugriffskosten auf den Externspeicher. Darüber hinaus ist es wichtig, dass die Operatoren nicht blockierend sind, d. h. , dass sie nicht erst die komplette Eingabe lesen müssen, bevor sie das erste Ergebnistupel berechnen können. Nur bei nicht blockierenden Operatoren können Pipelineeffekte innerhalb des Operatorbaums gut ausgenutzt werden.

Für die Laufzeittests haben wir ein praxisnahes Beispiel gewählt. An der Stanford Universität existiert ein Projekt von Gio Wiederhold, welches Daten über Schauspieler und Spielfilme sammelt und auf einer Webseite in Form von HTML-Tabellen zur Verfügung stellt. Die Adresse des Projekts lautet:

`http://www-db.stanford.edu/pub/movies/`

Aus Gründen der performanten Darstellung in Internetbrowsern sind die Daten einer Relation nicht in einer einzigen HTML-Tabelle zu finden, sondern sie sind über eine Reihe von HTML-Tabellen innerhalb einer einzigen HTML-Datei verteilt. Die in XXL implementierten Verfahren können mit einer solchen Struktur problemlos umgehen und führen selbstständig und effizient eine Vereinigung der Tabelleninhalte durch.

In den Tests wird eine HTML-Datei mit Schauspielern verwendet, die ca. 1 MB groß ist (6818 Tupel mit je 11 Attributen). Um die Größe der Datenmenge

variieren zu können, wird die Datei mehrfach hintereinander gehängt (1fach bis 32fach[1]).

Auf eine Validierung der Eingabedokumente wurde beim Einlesen verzichtet. Stimmt das Dokument nicht mit der Abbildung von XML nach Relational überein, so werden sowieso meistens entweder keine oder unkorrekte Tupel erzeugt. Die Validierung kann somit in den meisten Fällen auf der relationalen Seite erfolgen.

Der Einfluss von Externspeicherzugriffen wurde bei den Experimenten nicht gemessen. Dies liegt daran, dass die Eingangspuffer für beide Parser nicht durch Applikationen kontrollierbar sind. Hierdurch wären die Laufzeiten nicht aussagekräftig. Die Anzahl der Externspeicherzugriffe ist bei beiden Verfahren ohnehin identisch, da die Eingabedateien jeweils nur ein Mal sequentiell gelesen werden müssen. Bei den Experimenten wurden die XML Dateien daher komplett in den Hauptspeicher geladen und dann dem jeweiligen Parser als Datenstrom zur Verfügung gestellt.

Das Testszenario sieht so aus, dass die Einleseoperatoren abgearbeitet werden und bei jedem Tupel geprüft wird, ob es die korrekte Anzahl an Attributen besitzt. Die Messgröße war hier die Laufzeit, beginnend mit der Konstruktion des Cursors und endend mit der Rückgabe des letzten Tupels. Neben der Größe der Eingabedatei wurde beim SAX Operator auch die Größe des Kommunikationskanals variiert. Bei Kanalgröße eins wurde ein speziell optimierter Kanal verwendet, der ein wenig schneller ist, als der allgemeinere n-Tupel Kanal.

Im ersten Experiment wurde die optimale Kanalgröße ermittelt. Das Ergebnis ist in Abbildung 4.3 zu finden. Es zeigt sich, dass die Kanalgröße einen deutlichen Einfluss auf die Leistung hat. Begründet werden kann das damit, dass beim einelementigen Kanal nach jedem Tupel ein Threadwechsel erzwungen wird. Threadwechsel sind teuer und sollten wann immer möglich vermieden werden. Bei einem Kanal der Größe vier kann der Parser beispielsweise zunächst vier Tupel erzeugen, bevor der Threadwechsel wirklich nötig wird. Entsprechend kann der Einleseoperator vier Tupel hintereinander entnehmen,

[1] Der Wurzelknoten wird hierbei nicht vervielfacht, da ansonsten das erzeugte Dokument nicht mehr XML konform wäre.

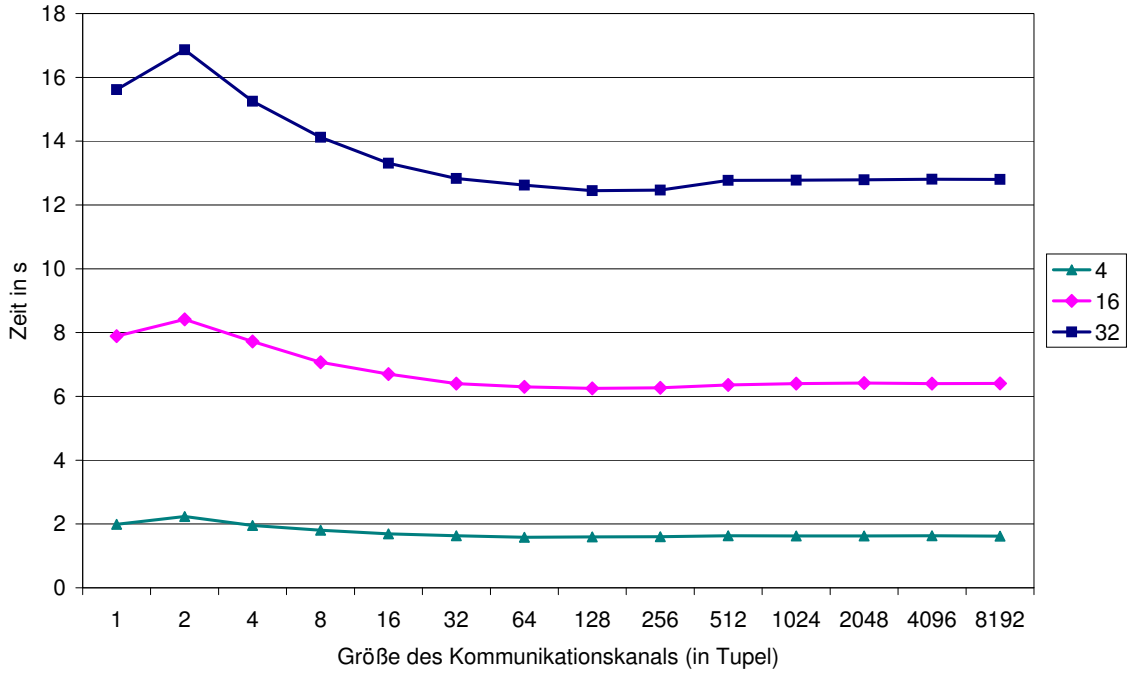


Abbildung 4.3: CPU-Zeit für die SAX Variante mit verschiedenen Größen des Kommunikationskanals (4, 16 und 32-fache Dateigröße)

ohne dass der Parser weiterlaufen muss. Auf das exakte Threadscheduling hat die Applikation jedoch keinen Einfluss - dies ist Sache von Java und vom Betriebssystem.

Offensichtlich kann die Erhöhung der Kanalgröße zu einer niedrigeren Anzahl an Threadwechseln führen, wodurch sich die Laufzeit verbessert. Ab einer gewissen Kanalgröße treten jedoch andere Effekte wie eine reduzierte Prozessorcachelokalität auf, so dass sich die Laufzeit dann wieder erhöht. Als optimale Kanalgröße wurden 128 Tupel ermittelt. Im Vergleich zu dem Ein-Tupel Kanal ergibt sich im Schnitt der Experimente ein Leistungsvorteil von 20,4%.

Deutlich zu erkennen ist außerdem, dass der Kommunikationskanal der Größe zwei langsamer ist, als der Kanal mit Größe eins. Dies liegt an der bereits erwähnten, speziell optimierten Implementierung des Ein-Tupel Kanals.

Das zweite Experiment (Abbildung 4.4) vergleicht die Performanz der beiden unterschiedlichen Operatorvarianten miteinander, wobei die DOM Variante gegen die schnellste SAX Variante mit einem Kommunikationskanal der Größe

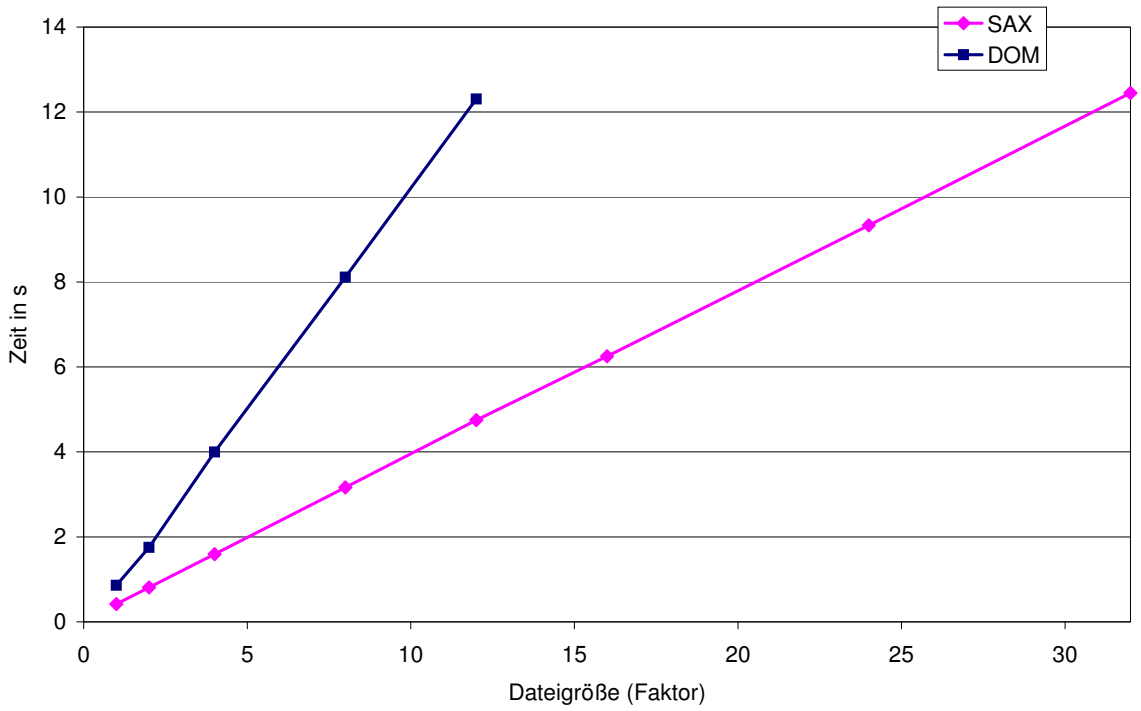


Abbildung 4.4: CPU-Zeit für DOM und SAX bei verschiedenen Dateigrößenfaktoren (Größe des Kommunikationskanals: 128)

128 Tupel antritt. Es zeigt sich, dass der Mehraufwand bei SAX für die Verwaltung des Kommunikationskanals deutlich geringer ist, als der Aufwand bei DOM für die Erstellung des Hauptspeicherbaums.

Der DOM-Hauptspeicherbaum besteht aus sehr vielen kleinen Objekten. Insgesamt belegen diese sehr viel Hauptspeicherplatz als die XML Datei selbst lang ist. Bei den Experimenten wurde der Java Laufzeitumgebung 256 MB Hauptspeicher zur Verfügung gestellt. Bei DOM traten Speicherüberlauffehler bereits bei der 16fachen Datei auf (etwa 16 MB). Bei diesem Experiment betrug die Größe des DOM-Baums also mindestens das 13fache der Größe der XML Datei. Für Server ist die DOM Variante daher generell nicht empfehlenswert. Bei vielen gleichzeitig ablaufenden Applikationen kann ein solcher Hauptspeicheraufwand nicht toleriert werden.

Im Schnitt schafft der SAX Einleseoperator 17.527 Tupel/s (2822 KB/s), während der DOM Operator sich mit 6.647 Tupel/s (1070 KB/s) begnügen muss. SAX ist somit etwa 2,64 Mal so schnell wie DOM.

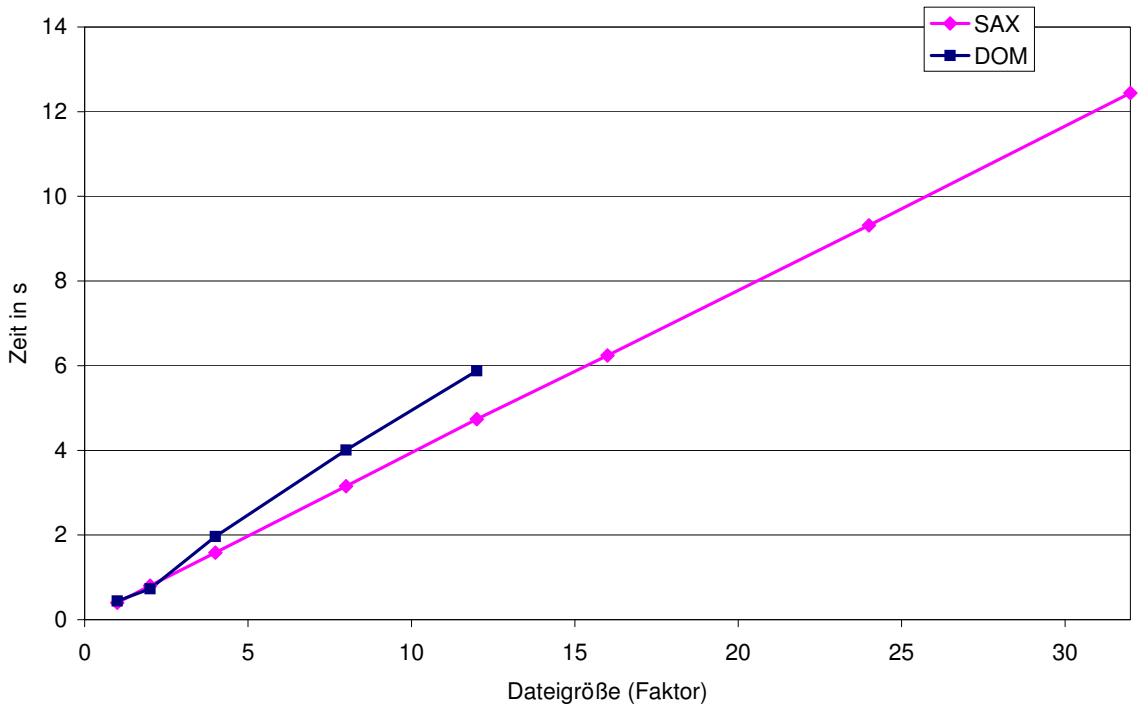


Abbildung 4.5: CPU-Zeit für DOM und SAX bei verschiedenen Dateigrößenfaktoren ohne die Zeiten für die Initialisierung (Größe des Kommunikationskanals: 128)

Ein weiteres Kriterium der Beurteilung ist die Blockierung des Operatorbaums durch den Einleseoperator. Hierzu lässt sich sagen, dass der DOM Parser auf jeden Fall erst das Dokument komplett gelesen haben muss, bevor er das erste Tupel generieren kann. Frühe Ergebnisse sind also nicht möglich. Dagegen kann die SAX-Variante ein Tupel bereits direkt nach dem Lesen der entsprechenden Dokumentstelle erzeugen.

Ein interessantes Bild zeigt sich, wenn man die Initialisierungszeit bei beiden Verfahren von der Gesamtlaufzeit abzieht (siehe Abbildung 4.5). Bei DOM fällt somit die Zeit für den Aufbau des Hauptspeicherbaums weg, währenddessen bei SAX nur wenige Millisekunden für die Konstruktion des Operators mit seinem Kommunikationskanal und dem Parserthread wegfallen.

Selbst in diesem Szenario ist SAX immer noch schneller als DOM. Der Grund hierfür liegt in der besseren Speicherlokalität der SAX Implementierung. SAX greift nur auf die Daten der XML Datei zu, währenddessen DOM den sehr

viel größeren Hauptspeicherbaum verwendet. Dadurch kann der Prozessorcache bei SAX besser ausgenutzt werden, was zu einer verbesserten Leistung führt. Selbst wenn ein DOM-Hauptspeicherbaum bereits vorliegt, ist es also effizienter, trotzdem SAX zu verwenden.

4.2 Zusammenfassung

Generell lässt sich sagen, dass datenzentrisches XML sehr effizient in das relationale Modell abgebildet werden kann. Datenraten von mehreren MB/s sind selbst auf nicht mehr ganz aktuellen Rechnern wie dem Testrechner (siehe Kapitel 2.5) möglich.

Die Anfragen auf relationalem XML erfolgen anschließend nicht in einer XML Anfragesprache, sondern sehr effizient in der Anfragesprache eines bewährten relationalen Datenbanksystems, beispielsweise in SQL.

Die Transformation von datenzentrischem XML kann prinzipiell mittels eines DOM oder eines SAX Parsers erfolgen. Das bedarfsgesteuerte Einlesen von relationalen XML Dateien erfordert bei Verwendung eines SAX-Parsers einige Klimmzüge. Der Aufwand für die Implementierung ist bei weitem höher als der Aufwand für die Implementierung der funktional identischen DOM Variante. Insgesamt lohnt sich dieser Mehraufwand jedoch, da Ressourcenverbrauch, die Tupelleistung und auch die nicht vorhandene Operatorbaumblockierung eindeutig für die SAX Variante sprechen.

Für den DOM-Operator spricht allerdings, dass mit dieser Variante sehr viel einfacher auch komplexe Konvertierungen von XML in das relationale Modell und zurück realisiert werden können. Ein SAX Parser setzt einen Eventhandler voraus, dessen Erstellung bei kompliziert strukturierten Daten sehr viel aufwendiger und fehleranfälliger ist, als wenn die Navigationsmethoden der DOM-API verwendet werden.

Kapitel 5

Nativer XML Speicher

Während es im vorigen Kapitel um relationales, also datenzentrisches XML ging, so dreht sich nun alles um dokumentenzentrisches XML, also um echt semi-strukturierte Daten. Abbildungen der Daten auf relationale Systeme sind hier zwar möglich, jedoch selten effizient.

Kernstück dieser Arbeit ist die Untersuchung von Methoden zur nativen Speicherung von XML Dokumenten. Solche Methoden dienen als Basis für native XML Datenbanksysteme. Bei echt semi-strukturierten Daten ist normaler Weise keine Grammatik für die betrachteten XML Dokumente vorhanden. Wenn dem doch so ist, so sind häufig Optimierungen in der Speicherungsschicht möglich [MWL+04].

Im Rahmen dieser Arbeit wurden ausschließlich Verfahren betrachtet, die ohne weitere Metadaten wie Grammatiken auskommen. Bereits hier war ein erhebliches Optimierungspotential im Vergleich zu Verfahren aus der Literatur vorhanden, das erst einmal ausgeschöpft wurde. In Zukunft könnte es interessant sein, die in dieser Arbeit optimierten Strukturen durch den Einsatz von Metadaten weiter zu verbessern.

In Kapitel 5.1 werden zunächst die typischen Eigenschaften von XML Datenbanksystemen skizziert. Anschließend (Kapitel 5.2) wird auf den Begriff der nativen XML Speicherungsstruktur eingegangen. Die Vorstellung des für die Arbeit wichtigsten Ansatzes aus der Literatur, den des nativen XML Datenbanksystems Natix [FHK+02], folgt in Kapitel 5.3. Dieser Ansatz ist wichtig

für das Verständnis von Kapitel 5.4, in dem die neu entwickelten Verfahren zur nativen XML Speicherung vorgestellt und genauer untersucht werden.

Kapitel 5.5 geht auf die Problematik von Indexstrukturen für nativen XML Speicher ein und stellt verschiedene Lösungsmöglichkeiten, sowie den implementierten und evaluierten Signaturindex vor. Zum Abschluss des Kapitels werden weitere Verfahren aus der Literatur erläutert und eingeordnet. Diesen Verfahren ist gemeinsam, dass sie experimentell nur unzureichend mit anderen Verfahren verglichen wurden. Dies mag unter anderem daran liegen, dass sich aus den verschiedenen, in der Literatur vorgeschlagenen Möglichkeiten zur Speicherung von XML noch kein Standardverfahren herauskristallisiert hat.

5.1 XML Datenbanksysteme

5.1.1 Eigenschaften von XML Datenbanksystemen

XML Datenbanksysteme dienen dazu, XML Dokumente zu speichern, Änderungen auf ihnen durchzuführen und Anfragen zu beantworten. Dies erfordert bereits die Unterstützung einiger grundlegender XML Standards.

Neben der vollen Implementierung des XML Standards vom World Wide Web Konsortiums ist mindestens die Anfragesprachen XPath (für Pfadanfragen) und eine weitere höhere XML Sprache mit Verbundoperationen, Aggregation und Transformation zu implementieren. Die meisten Systeme bieten hier XQuery und XSLT.

Doch diese Standards reichen bei weitem nicht aus, um die Funktionalität eines XML Datenbanksystems für Anwender nutzbar zu machen. Lange Zeit gab es keine deklarative Sprache für Einfüge- und Änderungsoperationen auf XML Dokumenten. Aus der Sicht eines Datenbanksystems waren für XML nur das Datenmodell und einige Anfragesprachen spezifiziert, jedoch keine *Datendefinitionssprache (DDL)*. Von einer *Datenmanipulationssprache (DML)* existierten nur Teile, welche die Anfragefunktionalität betrafen, nicht aber Standards für Einfüge-, Änderungs- und Löschoperationen. Dies stellt sich nun als ernstes

Problem heraus, weil ohne solche Standards die vielen, sich in Entwicklung befindlichen Systeme nicht untereinander kompatibel sein werden. Dies ist für einen Erfolg von XML Datenbanksystemen am Markt allerdings unbedingt notwendig.

Inzwischen scheint sich langsam *XML:DB* [Sta02] als Standard auszubilden. Jedoch ist XML:DB nur eine API, die zwar von Applikationsprogrammierern genutzt werden kann, allerdings ist sie nicht einmal für fortgeschrittene Datenbankanwender zu gebrauchen. Somit ist XML:DB nicht mit SQL vergleichbar. XML:DB enthält neben Operationen zum Einfügen und Löschen von kompletten XML Dokumenten auch Schnittstellen zu DOM oder SAX Applikationen. XML kann darüber hinaus auch als Text zurückgegeben werden. Als weiteren wesentlichen Bestandteil enthält XML:DB die deklarative Sprache *XUpdate*. Mit XUpdate können Einfüge-, Änderungs- und Löschoperationen auf einzelnen XML Dokumenten ausgeführt werden. Die Sprache XUpdate ist selbst in XML definiert und von der Syntax her an XSLT angelehnt. Hierdurch ist sie für Programmierer mit Vorkenntnissen im XML Bereich leicht erlernbar.

Unterstützt ein XML Datenbanksystem kein XML:DB, so müssen Applikationen auf andere Art und Weise angebunden werden. Neben JDBC für relationales XML empfiehlt sich die Unterstützung von DOM (genauer: virtuellem DOM) und SAX. Diese beiden Standards werden bislang von den meisten XML Applikationen verwendet. Im Fall von DOM müssen die Navigationsoperationen auf den Knoten sehr effizient durch das XML Datenbanksystem unterstützt werden (`getFirstChild`, `getNextSibling`, etc.).

5.1.1.1 Import und Export

Weitere wichtige Operationen sind Import und Export kompletter Dokumente. Dies muss performant realisiert sein, da diese Operationen häufig Verwendung finden. Wichtig ist außerdem, dass die Dokumente nach einem Durchlauf von Import und Export bytegenau aus dem System herauskommen. Hier darf kein Leerzeichen und kein Zeilenumbruch vergessen werden. Weiterhin besitzt die Reihenfolge von Knoten unterhalb eines Elternknotens in XML eine Bedeutung, die unbedingt berücksichtigt werden muss. Erste XML Systeme, die auf relationalen Datenbanksystemen basierten, hatten hiermit Probleme.

Da XML Datenbanksysteme häufig bei Internetanwendungen eingesetzt werden, ist die Einbindung von externen XML Quellen von besonderer Bedeutung. Aus dem selben Grund gehört die Zusammenarbeit von XML Datenbanksystem mit Web- und Applikationsserver zur Basisfunktionalität. Häufig werden entsprechende Serverprogramme bereits in das XML Datenbanksystem integriert.

Eine weitere Anbindung an XML Datenbanksysteme betrifft die so genannten *Webservices*. Webservices bieten im Internet bestimmte Dienste an, die interessant für die Anfrageverarbeitung in datenintensiven Systemen sind. Über diese Dienste können beispielsweise aktuelle Informationen wie Börsen- und Devisenkurse oder der Wetterbericht in Anfragen einfließen. Das den Webservices zugrunde liegende *Simple Object Access Protocol (SOAP)* [SOA04] ist XML basiert, so dass die Anbindung an XML Datenbanksysteme tiefer sein kann, als dies bei relationalen Datenbanksystemen strukturbedingt möglich ist.

5.1.1.2 Datenbankdienste

Außer der Unterstützung der genannten Standards, einer guten Performanz und Skalierbarkeit fordert der Anwender von XML Datenbanksystemen weitere Dienste. Diese sind bereits seit einigen Generationen von Datenbanksystemen bekannt. Die wichtigsten zu unterstützenden Dienste sind:

- Recovery
- Sicherheit (Rechteverwaltung)
- Sicherstellung der Datenintegrität
- Nebenläufigkeit
- Mehrbenutzersynchronisation

Recovery, Nebenläufigkeit und Mehrbenutzersynchronisation können durch einen Transaktionsmechanismus realisiert werden.

5.1.1.3 Indexstrukturen

Schon bei der Auswertung der einfachsten XPath Anfragen sind Indexstrukturen wichtig für die Leistung eines Systems. Allerdings beansprucht auch die Verwaltung von Indexstrukturen Zeit, weil Indexe bei Einfüge-, Änderungs- oder Löschoperationen aktuell gehalten werden müssen. Hier spielt also insbesondere das Verhältnis zwischen der Anzahl der baumverändernden Operationen zu der Anzahl der Suchoperationen eine wichtige Rolle.

Von Applikation zu Applikation kann dieses Verhältnis ganz unterschiedlich sein, was entsprechende Auswirkungen auf die Indexe und damit auf die Systemleistung hat. Somit muss eine Applikation die Möglichkeit besitzen, gewisse Indexstrukturen anzulegen und auf andere bewusst aus Leistungsgründen zu verzichten. Leider sind Indexstrukturen im XML Bereich zur Zeit noch weit entfernt von einer Standardisierung. Im Gegensatz zu SQL bei relationalen Systemen existiert bislang keine deklarative Sprache zur Definition von XML Indexstrukturen. Es sind noch nicht einmal Ansätze zur Definition einer entsprechenden Sprache zu finden.

Im Folgenden werden verschiedene Techniken vorgestellt, die zur Indexierung von XML verwendet werden können. Im XML Bereich existieren eine Reihe von Vorschlägen für verschiedene Indexstrukturen, die sich in fünf Klassen einteilen lassen:

- Knotenindex: In welchen Dokumenten ist ein Knoten mit einem vorgegebenen Namen n vorhanden?
- Pfadindex: In welchen Dokumenten ist ein bestimmter Pfad p vorhanden (von der Wurzel des Dokuments aus gesehen)?
- Strukturindex (für Verwandschaftsbeziehungen): In welchen der betrachteten Dokumente gibt es einen Knoten mit Namen n, der mit einem Knoten namens m in einer gewünschten Verwandtschaftsbeziehung steht (Vorfahre, Nachfahre, Elternknoten, Kindknoten, etc.)?
- Volltextindex: Da XML häufig im Dokumentenmanagement eingesetzt

wird, ist eine schnelle Textsuche wichtig. Hierfür können invertierte Listen und andere Techniken aus dem Bereich des Information Retrievals eingesetzt werden.

- Weitere inhaltsbezogene Indexe: Generell kann jede in der Literatur vorgeschlagene Indexstruktur auf XML Inhalte angewendet werden. Beispielsweise kann ein R-Baum [Gut84] auf den Geokoordinaten von in den Dokumenten gespeicherten Orten aufgebaut werden. Auch zusätzliche B^+-Bäume und ähnliche Strukturen sind für einige Zwecke interessant, beispielsweise für die Verfolgung von ID-IDREF-Verweisen.

Bei jeder dieser fünf Klassen sind zwei verschiedene Arten zu unterscheiden. Die erste Indexart gibt als Antwort zurück, in welchen Dokumenten Fundstellen existieren (wie oben beschrieben). Dies ist ausreichend, wenn der Hauptaufwand einer Anfrage darin besteht, die entsprechenden Dokumente zu lokalisieren. Diese Indexe reichen meistens aus, wenn mit kleinen Dokumenten gearbeitet wird. Die zweite Indexart, die so genannten *referenzierenden Indexstrukturen*, geben darüber hinaus die genauen Fundstellen in den Dokumenten zurück. Diese Indexart ist besonders wichtig, wenn wenige große Dokumente indexiert werden sollen. Durch die zu speichernden Referenzen sind diese Indexstrukturen i. d. R. auch sehr viel größer als nicht referenzierende Indexe. Voraussetzung für eine referenzierende Indexstruktur ist allerdings, dass sie überhaupt dazu in der Lage ist, Identifikatoren für beliebige Stellen in Dokumenten zu generieren. Dies macht eine enge Kopplung an die Speicherungsschicht notwendig, was bei der ersten Indexart nicht der Fall ist.

Als Identifikator für Stellen in Dokumenten eignen sich XPath-Ausdrücke im Allgemeinen nicht. Hierzu ein Beispiel: Ein Index hat `/site[1]/person[3]/name[1]` als Identifikator zurückgegeben. Direkt im Anschluss erfolgt eine Operation, die eine neue Person vor der dritten Person unterhalb von `site` im Dokument einfügt. Somit zeigt der ursprüngliche XPath-Ausdruck nun auf eine ganz andere Stelle im Dokument als zuvor, eventuell sogar auf eine Stelle, die niemals existiert hat. Hierdurch kann im Mehrbenutzerbetrieb die Antwort eines Indexes schon bei der ersten Benutzung des „Identifikators“ zur Lokalisation der falschen Stelle führen.

Im XML Bereich existiert nur noch eine weitere Möglichkeit für logische Identifikatoren, die ID/IDREF-Beziehungen. Einem Knoten wird in einem Attribut namens „ID“ ein Identifikator zugeordnet, der in dem Dokument eindeutig ist. Dieser Knoten kann dann referenziert werden, indem ein anderer Knoten ein Attribut namens „IDREF“ mit dem Identifikator als Wert erhält. Da XML Datenbanksysteme jedoch auch Dokumente mit bereits vorhandenen ID/IDREF-Beziehungen verarbeiten und diese auch identisch wieder herstellen können müssen, ist es nicht möglich, dieses Konzept direkt zu verwenden. Es muss also eine Lösung gewählt werden, die eng mit der Speicherungsstruktur verflochten ist. Dieses Problem wird in einem späteren Kapitel noch einmal aufgegriffen (Kapitel 5.5.1).

In der Literatur wurden bislang im Wesentlichen nicht referenzierende Indexstrukturen beschrieben. Bei den Strukturindexen wurden beispielsweise DataGuides [GW97] vorgeschlagen, die mit verhältnismäßig wenig Speicheraufwand die Menge aller Pfade eines Dokuments zusammenfassen. Hiervon existieren auch eine Reihe von neueren Varianten [CLO03, KSBG02].

Für strukturelle Indexe wurden weiterhin einige Techniken zur Knotennummerierung vorgeschlagen [HLM03]. Jedem Knoten eines Dokuments wird hierbei eine Folge von Zahlen zugeordnet. Eine Möglichkeit ist die Verwendung von Präorder- und Postorderreihenfolgewerten. Diese Werte geben an, in welcher Reihenfolge die Knoten bei einer Tiefensuche besucht werden, wenn man eine Prä- bzw. Postorder Traversierung durchführt [GKT03]. Aus den beiden Zahlen an jedem Knoten können dann direkt die wichtigsten Verwandtschaftsbeziehungen zwischen den Knoten abgelesen werden. Dies kann sehr einfach zur Beschleunigung der Auswertung von XPath-Ausdrücken eingesetzt werden.

Probleme gibt es bei den Nummerierungsschemata mit Einfügeoperationen und eventuell auch bei Löschoperationen. Bei durchgängiger Nummerierung ohne Freiräume hat eine einzelne Einfügeoperation zur Folge, dass alle Nummern des kompletten Dokuments neu berechnet werden müssen. Dies ist sehr zeitaufwendig und muss daher vermieden werden. Hierzu wurden einige Verfahren vorgeschlagen, die Bereiche zwischen zwei Nummern frei lassen [HLM03]. Somit kann die Nummerierung lange Zeit aufrecht erhalten werden, ohne dass das

ganze Dokument neu durchnummeriert werden muss. Der Nachteil bei Nummerierungsschemata mit Freiräumen ist jedoch, dass einige Verwandtschaftsbeziehungen nicht mehr direkt unterstützt werden können. Beispielsweise ist es möglich, dass bei Geschwisterknoten nicht ermittelt werden kann, ob zwischen ihnen ein weiterer Knoten liegt oder nicht.

In der Literatur existieren viele verschiedene Varianten, die mit den genannten Techniken arbeiten. Für die Diskussion von speziellen Indexierungstechniken wird an dieser Stelle auf die Literatur verwiesen [ABS99, LLD+02, RV03].

5.1.2 Native XML Datenbanksysteme

In vielen Artikeln taucht das Wort „nativ“ im Zusammenhang mit XML Datenbanksystemen auf. Das Wort „nativ“ ist geradezu ein Verkaufsargument für XML Datenbanksysteme. Doch was ist die genaue Definition für nativ? Hierfür existiert bislang keine eindeutige und konsensfähige Definition in der Literatur. Einige Fachbücher tragen den Begriff „nativ“ sogar im Titel, geben dann jedoch noch nicht einmal eine Definition hierfür an [BSM+03].

Ursprünglich geprägt wurde der Begriff von der Software AG, die ihr Tamino-System damit beworben hat [See03]. Unter nativen XML Datenbanksystemen wurden solche verstanden, die ganz auf XML ausgerichtet sind und die wichtigsten XML Standards unterstützen. Betont werden sollte damals besonders die Fähigkeit, XML Dokumente wieder 1:1, also auf das Byte genau, aus der Datenbank zurückgeliefert bekommen zu können.

Inzwischen wird jedoch mehr bzw. anderes mit dem Wort nativ verknüpft. Eine der am meisten zitierten Definitionen stammt von Tom Bradford [Bra04]: „Native XML databases (NXDs) are databases that store XML using an internalized format for faster overall processing, and representational flexibility. NXDs also provide support for indexing XML for improved query performance.“

Eine weitere Definition, die wie die vorherige ebenfalls auf XML.COM veröffentlicht wurde, widerspricht dem und sagt [Sta01]: „Is not required to have any particular underlying physical storage model. For example, it can be built

on a relational, hierarchical, or object-oriented database, or use a proprietary storage format such as indexed, compressed files."

Diese Definition macht jedoch die Unterscheidung zwischen nativen XML Datenbanksystemen und XML befähigten Systemen nahezu unmöglich. In der Praxis werden jedoch diese beiden Begriffe für unterschiedliche XML Systeme verwendet. In der vorliegenden Arbeit soll daher mit den folgenden, neu entwickelten Definitionen gearbeitet werden:

- **XML befähigtes Datenbanksystem:** Ein Datenbanksystem, dem eine relationale, objektrelationale, objektorientierte oder hierarchische Verarbeitung zugrunde liegt. Das System muss die Speicherung von XML, sowie mindestens eine Anfragesprache unterstützen.
- **XML Datenbanksystem:** Ein Datenbanksystem, das XML Dokumente bytegenau verwaltet und mindestens die Funktionalität des XML:DB-Standards anbietet. Darüber hinaus ist die Verwaltung von XML Grammatiken und eine Validierungseinheit verpflichtend.
- **Natives XML Datenbanksystem:** Ein XML Datenbanksystem, das zur Speicherung der Daten spezielle, strukturerhaltende Verfahren anbietet (so genannter nativer XML Speicher). Die Speicherung von Dokumenten darf nicht grundsätzlich als Ganzes erfolgen, und Dokumente dürfen nicht ausschließlich in Relationen bzw. ähnlichen Strukturen abgelegt werden.

Etwas problematisch ist, dass eines der ersten industriellen XML Datenbanksysteme nach diesen Definitionen nicht mehr als natives System zu bezeichnen ist. Tamino [Sch03] speichert XML Dokumente mittlerweile grundsätzlich als Ganzes ab.

5.1.3 XML Datenbanksysteme in der Praxis

Der Markt für XML Datenbanksysteme ist momentan ein wenig unübersichtlich, da sehr viele Projekte an Universitäten zur Gründung von spin-off Firmen

geführt haben. Wirklich produktiv einsetzbar sind momentan allerdings die wenigsten dieser Systeme. Marktführer im Bereich der XML Datenbanksysteme ist die Software AG mit ihrem Tamino XML Server [Sch03, Sch01, SW00]. Die großen Datenbankfirmen Oracle, IBM und Microsoft engagieren sich in diesem Markt bislang noch nicht direkt, integrieren in ihre großen objektrelationalen Systeme allerdings immer mehr Merkmale von XML Datenbanken. Somit sind ihre Systeme als XML befähigt zu bezeichnen. Es ist sicherlich nur eine Frage der Zeit, bis nativer XML Speicher auch hier Einzug erhält.

Den industriell vertriebenen XML Datenbanksystemen ist gemeinsam, dass über ihre internen Abläufe wenig bekannt ist. Die internen Strukturen werden als Wettbewerbsvorteil gegenüber der Konkurrenz angesehen und werden deshalb geheim gehalten. Bei open-source Systemen wie Apache Xindice [Xin04] oder eXist [eXi04] sieht dies natürlich anders aus. Durch Studien des Quelltextes oder Publikationen auf den Webseiten der Projekte sind hier mehr Details zu erfahren. Über die aus der Forschung stammenden Systeme wurde i. d. R. in wissenschaftlichen Kreisen publiziert, so dass hier am meisten über die verwendeten Konzepte und deren Fundierung bekannt ist. Zu dieser Gruppe von Systemen gehören Natix [FHK+02, FKM01], OrientStore [MLLA03], TIMBER [JAKC+02, PAKC+03] und der Xyleme Zone Server [ACFR02, ACV+02, Xyl01].

Von Bedeutung ist ebenfalls die Bibliothek Berkeley DB XML [LW04], auch wenn sie nicht unter den Oberbegriff eines „Systems“ fällt. Berkeley DB XML basiert auf dem häufig verwendeten Berkeley DB für relationale Datenbanksysteme und erweitert dies um Funktionalität für XML Datenbanksysteme. Mit Berkeley DB XML können auf einfache Art und Weise Systeme erstellt werden, wobei das komplette Datenbanksystem dann in die Applikation eingebettet wird. Dies hat Vorteile im Administrationsaufwand, wenn auf einem Server nur einzelne Applikationen ein XML Datenbanksystem verwenden. Wenn jedoch viele Applikationen persistente XML Speicherung benötigen, so ist von Berkeley DB XML abzuraten, weil alle Applikationen durch das Einbetten des Datenbanksystems stark aufbläht werden.

5.2 Speicherungsverfahren für XML

In der Speicherungsebene einer XML Datenbank werden die vollständigen Daten der XML Dokumente gespeichert. Die Speicherungsebene ist grundsätzlich zu unterscheiden von der darüber liegenden Indexierungsebene. In den Indexen können zwar auch Daten der Dokumente auftauchen, diese reichen jedoch i. Allg. nicht zur bytegenauen Wiederherstellung der Dokumente aus. Typisch für die Indexierungsebene sind weiterhin Verweise auf die eigentlichen Daten in der Speicherungsebene, allerdings ist dies kein notwendiges Kriterium für die Indexierungsebene.

Eine saubere Trennung von Speicherung und Indexierung ist teilweise problematisch. In objektrelationalen Datenbanksystemen werden i. d. R. mehrere Datensätze auf einen Externspeicherblock abgebildet. Die Adresse eines Datensatzes, der so genannte Tupelidentifikator bestehend aus Blocknummer und einem Identifikator innerhalb des Blocks, bleibt konstant, auch dann, wenn Änderungen erfolgen oder neue Daten eingefügt werden.

Dies hat jedoch nicht nur Vorteile. In einem Beispielszenario führt eine Applikation sehr häufig eine Bereichssuche auf den Nachnamen von Personen durch. Dies liegt hierbei daran, dass eine Bildschirmmaske immer 25 Personen auf einmal in einer Liste auf dem Bildschirm anzeigt. Führt man diese Suche auf einer Struktur mit B^+-Baum Index aus, der Verweise auf die eigentlichen Datensätze enthält, so werden durch diesen zunächst alle Datensatzidentifikatoren mit passendem Personennamen ermittelt (Bereichssuche). Anschließend werden von einer Komponente der Speicherungsschicht die 25 dazu passenden Datensätze gelesen, was zu etwa 25 Seeks führen wird, wenn die Datensätze zu den Personen nicht zufällig direkt nacheinander in das System eingefügt wurden.

In diesem Beispiel kann eine Speicherung mit Clusterbildung (in Oracle: Index-organized Table [Ora04]) helfen. Hier werden die Daten in den Blättern des Indexes (dem B^+-Baum) abgelegt. Die Bereichsanfrage kann nun komplett aus dem Index beantwortet werden. Der Vorteil dieses Verfahrens ist, dass sehr wahrscheinlich immer mehrere Datensätze innerhalb eines Blattes liegen, und hierdurch die Anzahl der benötigten Externspeicherzugriffe gering ist.

Weitere vorhandene Indexe können nun allerdings nicht mehr direkt auf die physikalische Position des Datensatzes verweisen, weil sich diese bei jedem Split in der Indexstruktur verändern kann. Zusätzliche Indexstrukturen können somit nur noch auf den Schlüsselwert verweisen, mit dem der Datensatz in der Struktur wiedergefunden werden kann.

Eine Index-organized Table ist also Indexstruktur und Speicherungsstruktur zugleich. Solche integrierten Indexstrukturen in der Speicherungsebene sind auch im XML Bereich zu finden, wie später noch erläutert wird.

5.2.1 Grundannahmen

Die Speicherungsverfahren für XML unterscheiden sich ganz grundlegend dadurch, welche Grundannahmen getroffen werden. Eine extreme Grundannahme ist, dass ein System sehr viele kleine XML Dokumente verwaltet (Millionen von Dokumenten, die beispielsweise kleiner als 4 KB sind). Hier kann ein einzelnes XML Dokument mit der Rolle eines Tupels in relationalen Systemen verglichen werden, und zwar sowohl hinsichtlich der Größe als auch hinsichtlich des Inhalts. Das andere Extrem ist die Verwaltung von sehr wenigen oder sogar von nur einem einzigen großen Dokument. Hierbei nimmt ein XML Dokument die Rolle einer kompletten Datenbank ein.

Bei vielen, kleinen Dokumenten ist eine Speicherung als Ganzes nahe am theoretischen Optimum. Jedes Dokument passt hier i. d. R. in einen einzigen Block. Dokumente können mit einem einzigen Seeks gelesen werden und DOM im Hauptspeicher ist für die Ausführung der wichtigsten Operationen auf den kleinen Dokumenten absolut ausreichend. Für diesen Fall ist somit recht offensichtlich, wie die Speicherungsschicht aufgebaut sein muss. Deshalb soll in dieser Arbeit die Speicherung von vielen kleinen Dokumenten nicht weiter betrachtet werden.

Im Fall von wenigen, großen Dokumente wäre eine Speicherung als Ganzes äußerst ineffektiv, da bei jeder kleinsten Änderung ein ganzes Dokument eingelesen[1], bearbeitet und anschließend wieder weggeschrieben werden müsste.

[1] Dieses wäre bei sehr großen Dokumenten mit DOM schon gar nicht mehr möglich.

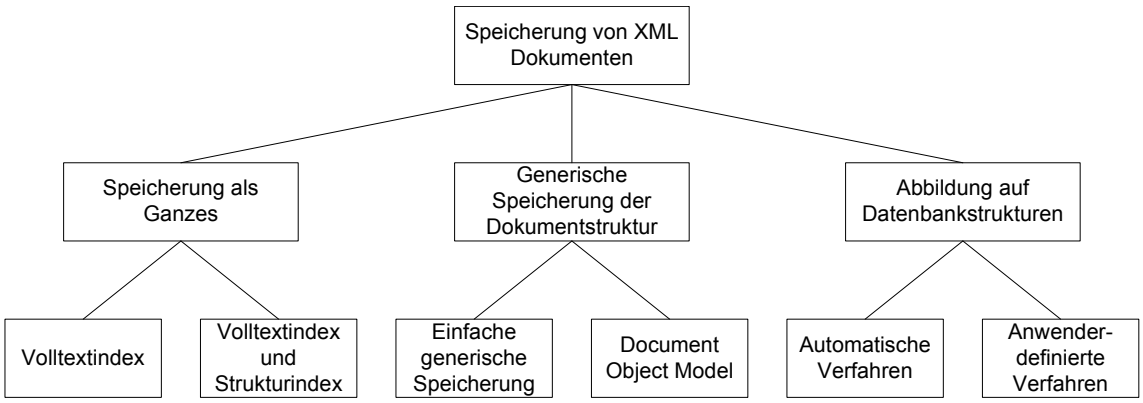


Abbildung 5.1: Klassifikation von Verfahren zur Speicherung von XML nach [KM03]

Hierfür sind also Speicherungsstrukturen notwendig, welche die Dokumente in kleineren Einheiten ablegen.

Für die Praxis ist interessant, bis zu welcher mittleren Dokumentgröße die Speicherung als Ganzes einer nativen Speicherung vorzuziehen ist. Im Rahmen dieser Arbeit werden in diesem und im folgenden Kapitel hierzu einige Aussagen gemacht.

5.2.2 Klassifikation von Speicherungsverfahren

Im folgenden soll eine kurze Übersicht über verschiedene Techniken zur XML Speicherung gegeben werden. Hierzu ist es hilfreich, sich an einer Klassifikation von Speicherungsverfahren zu orientieren. Leider gibt es in der Literatur bislang nur wenige Klassifikationen, die einigermaßen vollständig sind.

Die Klassifikation von Klettke und Meyer [KM03] ist zweistufig. Sie ist in Abbildung 5.1 dargestellt. Auf der ersten Ebene wird zwischen der bereits beschriebenen Speicherung als Ganzes, einer generischen Speicherungsstruktur und der Abbildung auf Datenbankstrukturen unterschieden. Diese drei Gebiete werden dann in der zweiten Ebene weiter untergliedert.

Bei der Speicherung als Ganzes sind Indexstrukturen für die Leistungsfähigkeit sehr wichtig. Hier unterscheidet die gegebene Klassifikation zwischen Art und Anzahl der verwendeten Indexe.

Der zweite Punkt, die generische Speicherung, umfasst alle Modelle, die versuchen, die Struktur eines Dokuments in eigene, speziell entwickelte Strukturen abzubilden. Dies kann entweder anhand einer Abbildung von DOM-Strukturen erfolgen oder es können andere, strukturerhaltende Konzepte umgesetzt werden. Dieser Punkt entspricht also den bereits angesprochenen nativen Methoden zur Speicherung von XML, bei denen die Baumstruktur von XML eine spezielle Berücksichtigung findet. Hierdurch kann auf bestimmte Teile eines Dokuments gezielt zugegriffen werden. Dies ist sowohl für die Leistung von Anfragen als auch für Einfüge-, Änderungs- und Löschoperationen essentiell. Die in dieser Arbeit genauer untersuchten Verfahren fallen hauptsächlich in diese Kategorie.

Auch beim dritten Punkt, der Abbildung auf Datenbankstrukturen, kann DOM als Vorbild dienen. Allerdings werden die DOM-Knoten hier in Relationen oder ähnliche Strukturen der Datenbank umgesetzt. Dies kann entweder automatisch erfolgen oder vom Anwender gesteuert werden. Auf diese relationalen Speicherungsstrukturen für XML wird noch genauer in Kapitel 6 eingegangen.

5.2.3 Kompression

Ein weiterer wichtiger Punkt bei der Speicherung von XML ist die Kompression. XML Dokumente sind Textdateien, deren Datenmenge sich typischer Weise sehr gut durch Kompressionsverfahren reduzieren lässt [LS00, CN04]. Kompression lässt sich besonders effektiv anwenden, wenn die einzelnen Teile eines Dokuments nicht zu klein sind. Daher ist Kompression gut bei der Speicherung als Ganzes und eingeschränkt bei der generischen Speicherung anwendbar. Ein Nachteil von Kompressionsverfahren ist jedoch, dass der Zugriff auf Strukturen innerhalb eines komprimierten Blocks im Normalfall die komplette Dekomprimierung des Blocks nötig macht. Dies ist sehr zeitaufwendig. Somit existieren bei der Kompression von XML Dokumenten zwei sich widerstrebende Optimierungskriterien:

- Kompression erreicht nur dann eine gute Kompressionsrate, wenn **größere** Einzelteile als Block bearbeitet werden.

- Für den Zugriff auf vorgegebene Stellen innerhalb von Dokumenten empfehlen sich **kleine** Einzelteile, da dann weniger nicht benötigte Abschnitte des Dokuments ausgepackt werden müssen.

5.3 XML Speicherung in Natix

An dieser Stelle soll die Technik einer nativen Speicherungsstruktur am Beispiel von Natix vorgestellt werden. Natix [FHK+02] ist ein natives XML Datenbanksystem, das am Lehrstuhl Praktische Informatik III in Mannheim unter der Leitung von Prof. Dr. Guido Moerkotte entwickelt wurde. Kernstück dieses Systems ist der native Speicher [KM99,KM00]. Die Implementierung von darauf aufsetzenden Datenbankdiensten ist in Natix bereits sehr weit fortgeschritten, so dass das System nahezu den Status eines einsatzbereiten Produktes hat.

Die Speicherungsstruktur von Natix [KM00] speichert Teilbäume eines Dokuments als Records in einem Recordmanager ab. Jedes Record enthält genau einen Teilbaum mit genau einer Wurzel. Hierdurch soll eine gewisse Lokalität erreicht werden, die besonders bei Navigationsoperationen ausgenutzt werden kann. Beispielsweise werden hierdurch `getChildNode` und `getNextSibling` aus der DOM-API sehr effizient unterstützt.

Das Grundmodell von Natix erinnert zunächst stark an die DOM-API. Es existieren Datentypen für Knoten, Attribute, Text und andere XML Elemente. Darüber hinaus werden Datentypen für die Verbindung von Teilbäumen spezifiziert. Hierzu existieren so genannte Proxy- und Scaffoldknoten. Ein *Proxyknoten* enthält einen Zeiger auf ein Record, das einen Teilbaum enthält. Logisch gesehen wird dieser Teilbaum anstelle des Proxyknotens in das Dokument eingesetzt (Proxyknoten = Stellvertreterknoten). Ein *Scaffoldknoten* hat ebenfalls keinen Einfluss auf die logische Struktur eines XML Dokuments (beide Knotenarten werden daher im Folgenden auch virtuelle Knoten genannt). Ein Scaffoldknoten (=Gerüstknoten) fächert nur auf, d. h. logisch gesehen sind die Kindknoten eines Scaffoldknotens s die Kindknoten des nächsten, nicht virtuellen Elternknotens von s. Ein Beispiel für ein über mehrere Records verteiltes XML Dokument befindet sich in Abbildung 5.2.

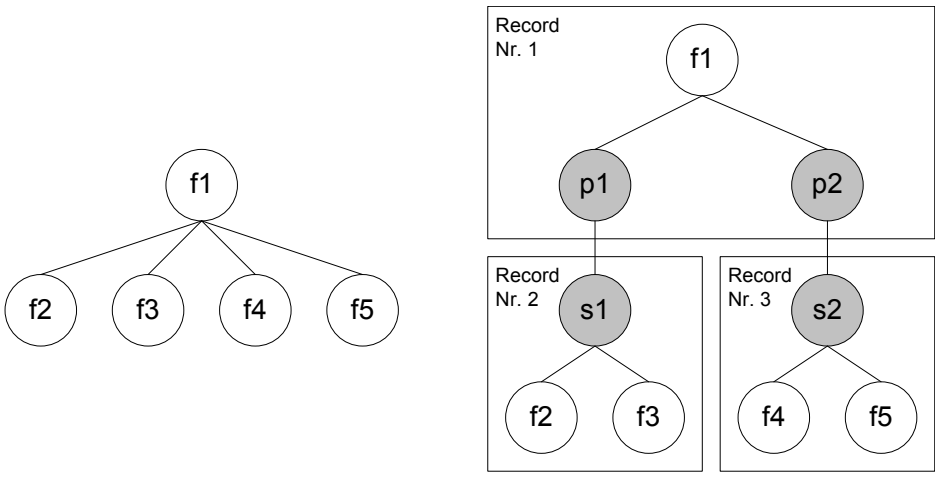


Abbildung 5.2: Ein logisches XML Dokument (links) und seine Natix-Repräsentation in drei Records (rechts)

Es sei noch bemerkt, dass es in dieser Grundstruktur möglich ist, von einem Kindknoten zum Elternknoten hoch zu navigieren. Hierfür enthält jeder Teilbaum den Recordidentifikator des Elternrecords. Für viele Operationen in DOM und XPath ist die Möglichkeit, effizient zum Elternknoten gehen zu können, für eine gute Performanz notwendig.

Die Serialisierung eines Teilbaums in einen Bytestrom erfolgt mittels eines speziellen Converters, der die Byterepräsentation der Records relativ klein hält. Trotz der im Vergleich zum rein logischen XML Dokument zusätzlich benötigten Informationen (Proxyknoten mit Identifikatoren, Scaffoldknoten) liegt der benötigte Speicher für ein Dokument i. d. R. nur wenig über dem Speicherplatzbedarf der XML Datei.

Die wichtigsten Operationen auf dieser Grundstruktur sind die Suche nach Knoten, sowie Einfüge-, Änderungs- und Löschoperationen. Bei der Suche eines Knotens mit XPath wird das Dokument von der Wurzel ausgehend in einer Tiefensuche traversiert. Jeder hierbei betrachtete Knoten wird daraufhin untersucht, ob er den gegebenen XPath-Ausdruck erfüllt. Ist dies der Fall, wird die gefundene Stelle entweder einer Prozedur gemeldet, die dann Aktionen ausführen kann, oder über einen Kanal an einen Cursor übergeben (ähnlich wie in Kapitel 4.1.2). Erfüllt der Knoten die XPath-Bedingung nicht, so wird geprüft, ob sich unterhalb des Knotens überhaupt Ergebnisse befinden können.

Ist dies nicht der Fall, so wird auf die Traversierung des Teilbaums unterhalb des Knotens verzichtet. Bei einer Suchoperation muss also nicht das komplette XML Dokument gelesen werden, wie es bei Systemen der Fall ist, die XML Dokumente als ganzes abspeichern, sondern es kann zielgerichtet von Record zu Record navigiert werden.

Bei Einfüge-, Änderungs- und Löschoperationen wird zunächst eine Suchoperation durchgeführt, die den Knoten lokalisiert, auf dem die jeweilige Operation durchgeführt werden soll. Bei einer Einfügeoperation ist der gefundene Knoten der Elternknoten, in den an einer bestimmten Stelle ein Knoten oder ein Teilbaum eingefügt werden soll. Bei Änderungs- und Löschoperationen ist der gefundene Knoten bereits derjenige, auf den sich die Operation bezieht.

Wenn eine Stelle im Dokument lokalisiert wurde, so erfolgt die eigentliche Operation innerhalb einer Prozedur, die den Teilbaum anschließend wieder im Recordmanager speichert. Hierbei können jedoch Über- und Unterläufe auftreten, die gesondert behandelt werden müssen.

Unterläufe kommen in der Praxis relativ selten vor, weil die typische Anfragelast in XML Datenbanksystemen viele Einfüge- und Änderungsoperationen, jedoch wenige Löschoperationen enthält. Auf Unterläufe könnte man mit der Vereinigung von Teilbäumen reagieren, wenn die Größe eines Records unter einen gewissen minimalen Wert abfällt. Dies wurde jedoch weder in der Literatur, noch in dieser Arbeit untersucht. Die Behandlung von Unterläufen ist auch generell problematisch. Bei gewissen Dokumentstrukturen lassen sich kleine Records nicht vermeiden. Eine Unterlaufbehandlung muss an solchen Stellen im Dokument erkennen, dass eine Zusammenlegung der Teilbäume nicht möglich ist. Wenn sie dies nicht erkennt, entsteht unter Umständen ein hoher Aufwand für erfolglose Vereinigungsoperationen.

Bei einem Überlauf wird der aktuelle Teilbaum mit einem Splitalgorithmus in zwei oder mehr kleinere Teilbäume aufgeteilt. Beim *Natix-Splitalgorithmus* wird der Ausgangsbaum in drei disjunkte Partitionen aufgeteilt: eine linke Partition, eine rechte Partition und ein Separatorteilbaum S, der die linke und die rechte Partition voneinander abtrennt (siehe Abbildung 5.3.b). Ein solcher Split wird eindeutig durch den so genannten Splitknoten d beschrieben. Dieser Kno-

ten gehört per Definition immer zur rechten Partition. Die Knoten oberhalb von d (seine Vorfahren) gehören zum Separator, währenddessen alle anderen Knoten, die in Dokumentreihenfolge vor dem Splitknoten liegen und nicht bereits zum Separatorteilbaum gehören, der linken Partition zugeschrieben werden.

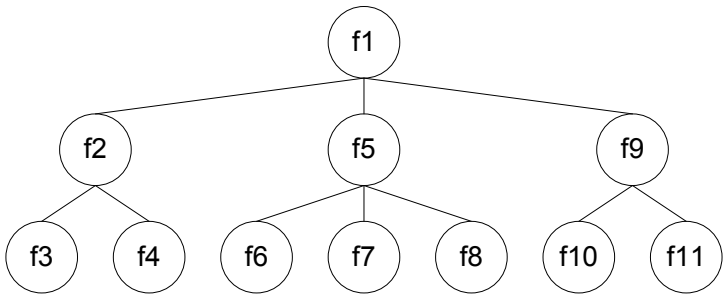
Der Knoten d wird beim Natix-Split so gewählt, dass er ein vorgegebenes Kriterium möglichst gut erfüllt. Normaler Weise wird versucht, dass die linke und die rechte Partition eine ähnliche serialisierte Größe besitzen.

Nach dem erfolgten logischen Split müssen die Teilbäume der linken und rechten Partition als Records in den Recordmanager eingefügt werden (siehe Abbildung 5.3.c). In jeder Partition werden dazu zunächst diejenigen Teilbäume durch Scaffoldknoten zusammengefasst, die denselben direkten Elternknoten im Separatorteilbaum besitzen (siehe Record Nr. 4 in Abbildung 5.3.c). Anschließend besteht jede Partition aus mindestens einem, höchstens $h(S)$ Teilbäumen, die in Records serialisiert und persistent gemacht werden. Nicht vergessen werden darf hierbei, dass in den durch die Teilbäume referenzierten Kinderrecords die Identifikatoren der Elternrecords auf den neuen Stand gebracht werden müssen.

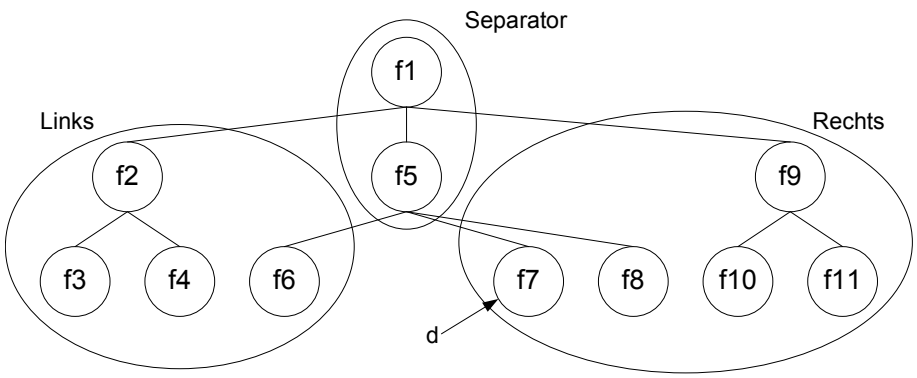
In den Separatorteilbaum müssen dann mindestens zwei, höchstens $2 * h(S)$ Proxyknoten neu eingefügt werden. Hierbei zeigt sich deutlich, dass ein Split in zwei gleichgroße Partitionen L und R nicht unbedingt zu zwei Records und auch nicht zu mehreren in etwa gleichgroßen Records führen muss. Der Separatorknoten wird anschließend in den Elternknoten eingefügt. Wenn dieser überläuft, ist wiederum ein Split durchzuführen.

Bei allen drei Arten von Operationen bleiben die nötigen Änderungen am Dokument auf die Records entlang des Pfades vom Wurzelrecord zur Stelle der Änderung begrenzt. Über die Anzahl der nötigen Externspeicherzugriffe kann leider wenig ausgesagt werden, weil die Höhe des Separatorbaums nur durch $|R| - 1$ begrenzt ist. Es gibt einen etwas hypothetischen Fall eines Teilbaums bestehend aus drei Knoten, bei dessen Split genau drei Externspeicherzugriffe auftreten. Baut man sich einen Teilbaum aus $|R|$ Knoten auf, so kann man immer einen Fall konstruieren, in dem mehr als $\frac{2}{3}|R|$ Externspeicherzugriffe benötigt werden. Somit liegt der schlechteste Fall für einen Split in der Größenordnung von $O(|R|)$ Externspeicherzugriffen. In der Praxis zeigt sich jedoch,

a) Logischer Ausgangsbaum



b) Aufteilung in logische Teilbäume



c) Aufteilung in physische Teilbäume

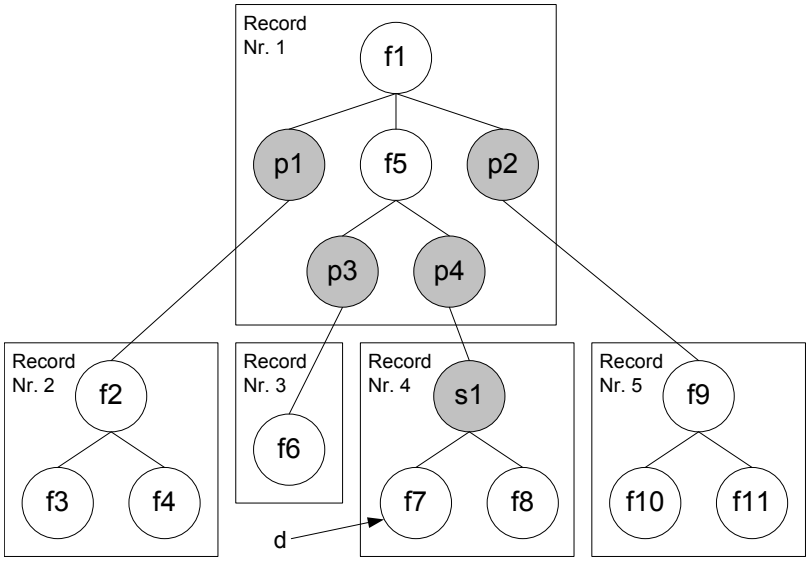


Abbildung 5.3: Aufteilung eines XML Teilbaums nach Natix (Splitalgorithmus)

dass die Anzahl der Externspeicherzugriffe klein gegen die Anzahl der Knoten in einem Teilbaum ist. Somit spielt die theoretisch hergeleitete Obergrenze in der Praxis keine Rolle.

Durch den Natix-Splitalgorithmus, im Folgenden auch *Separator Split* ***(SepS)*** genannt, wird die vorgestellte XML Speicherungsstruktur zu einem *grow-and-post tree* [Lom91], was die Recordstruktur angeht. Die wesentliche Eigenschaft so eines Baumes ist, dass alle bei der Wurzel beginnenden Pfade dieselbe Länge (hier: dieselbe Anzahl an Records) bis zum Erreichen der untersten Baumebene besitzen. Der Baum ist also höhenbalanciert.

Diese Eigenschaft hat die Natix-Speicherungsstruktur mit einem B-Baum gemeinsam. Man könnte die Natix-Speicherungsstruktur also auch als B-Baum der nativen XML Datenbanksysteme bezeichnen, da noch weitere Gemeinsamkeiten bestehen: Beide legen im Unterschied zum B^+-Baum die Daten sowohl in den Blättern, als auch in den inneren Baumknoten ab. Somit benötigen Suchvorgänge ganz unterschiedlich viele Externspeicherzugriffe, je nachdem, in welcher Tiefe des Baumes sich die gesuchte Information befindet.

Ein wesentlicher Faktor für die Anzahl der Zugriffe bei einem Suchvorgang ist außerdem die Höhe des XML Dokuments. Über diese Höhe kann allerdings nur sehr wenig ausgesagt werden. Je nach der Art der Daten kann die Höhe sehr unterschiedlich sein. Auch innerhalb eines Dokuments kann die Höhe stark variieren. Die Auswirkungen hiervon sollen an einem kurzen Beispiel illustriert werden. Nehmen wir an, dass in einem Dokument ein Pfad von der Wurzel bis zu einem Blatt vorhanden ist, der aus insgesamt 100 Knoten besteht. Die vielen anderen Pfade des Dokuments (von der Wurzel bis zu einem Blatt) besitzen dagegen nur eine Höhe von maximal zehn. Weiterhin wird angenommen, die Natix-Struktur besitzt eine Höhe von $h = 10$ Records. Bei höhenbalancierten Bäumen gilt, dass die meisten Records sich auf der untersten Baumebene befinden. Daraus lässt sich folgern, dass die meisten Blattknoten der Pfade p mit $|p| \leq 10$ sich in der untersten Recordebene befinden müssen. Die Natix-Struktur passt sich also nicht gut an die tatsächlichen Höhenverteilungen in XML Dokumenten an.

Die zunächst vorhandene Höhenbalancierung der Natix-Struktur wird in der Praxis allerdings von Löschoperationen wieder zerstört. Wenn nach Löschun-

gen Teilbäume auftreten, die nur aus einem Proxyknoten bestehen (oder aus einem Scaffoldknoten, gefolgt von einem Proxyknoten), so werden die kompletten Teilbäume gelöscht und die Zeiger entsprechend umgebogen.

In Natix existiert noch eine zweite Variante dieses Splitalgorithmus, der so genannte *Matrix Split*. Er setzt auf dem Separator Split auf und nutzt eventuell vorhandene Meta-Informationen über die Dokumentstruktur aus. Mittels einer Matrix kann festgelegt werden, welche Knoten mit anderen Knoten nach Möglichkeit in Teilbäumen zusammengehalten werden sollen. Außerdem kann das Gegenteil hiervon festgelegt werden, also welche Knotenarten nicht zusammen gespeichert werden sollen. Die Splitmatrix schränkt also die Wahlfreiheit des Separator Split Algorithmus ein. Durch die Splitmatrix werden einige Schnitte im Baum bevorzugt und andere verhindert. Allerdings kann sich der Splitalgorithmus letzten Endes immer über den Rat der Matrix hinwegsetzen und die Schnitte im Baum so platzieren, wie er es in der aktuellen Situation für richtig hält. Dies ist in einigen Situationen zwingend erforderlich, nämlich immer dann, wenn nach dem Rat der Matrix überhaupt kein Split durchgeführt werden dürfte.

Eine Splitmatrix kann beispielsweise aus einer vorhandenen Grammatik zu einem Dokument automatisch erzeugt werden. Auf eine solche Erstellung der Matrix und den dazu verwendbaren Algorithmen gehen die Autoren von Natix jedoch nicht ein. Da der Matrix Split zusätzliche Meta-Informationen benutzt, die bei echt semi-strukturierten Daten i. d. R. nicht vorliegen, wird auf eine weitere Betrachtung des Matrix Splits im Folgenden verzichtet.

5.4 XML Speicherung in XXL

Für diese Arbeit wurde in XXL im Paket `xxl.core.xml.storage` ein nativer XML Speicher nach dem Vorbild von Natix implementiert. Er basiert auf den in Kapitel 3 vorgestellten, erweiterten Basiskomponenten von XXL, so dass die eigentliche Implementierung der Speicherungsstruktur mit einem einigermaßen überschaubaren Aufwand möglich war. Die generelle Objekthierarchie für die Speicherung ist identisch mit der in Kapitel 3 in Abbildung 3.20 beschriebe-

nen. Die Umwandlung eines Records in einen XML Teilbaum übernimmt ein spezieller Converter. Diese Operation ist in Java relativ zeitaufwendig, weshalb sie zu vermeiden ist wenn möglich.

Das Gerüst in XXL dient der Untersuchung von verschiedenen neuartigen Konzepten, die für die native XML Speicherung entwickelt wurden. Die Anbindung der Splitalgorithmen wurde, soweit es möglich ist, allgemein gehalten, damit ganz verschiedene Verfahren im selben Gerüst implementiert und miteinander verglichen werden können.

In den folgenden Abschnitten geht es zunächst um einige Teilprobleme, beispielsweise um die Wahl der exakten Einfügeposition. Danach werden neu entwickelte Splitalgorithmen vorgestellt. Es folgt die Beschreibung einiger Details der Implementierung und abschließend wird eine Evaluierung mittels des XML Benchmarks XMark [SWK+02] vorgenommen.

5.4.1 Die Auswahl der exakten Einfügeposition

Durch die Einführung von virtuellen Knoten in Natix stellt sich ein neues Problem. Durch virtuelle Knoten kann es in der physischen Repräsentation eines Dokuments verschiedene Stellen geben, die logisch gesehen äquivalent sind. Somit gibt es beim Einfügen eines neuen Knotens Wahlmöglichkeiten in der exakten physischen Position. Im Beispiel in Abbildung 5.4 soll ein Knoten unterhalb von Knoten $f1$ eingefügt werden, der logisch gesehen unmittelbar auf den Knoten $f2$ folgen soll. Hierfür gibt es in der physischen Struktur vier verschiedene Möglichkeiten, die in der Abbildung mit einem Kreuz markiert sind. Logisch gesehen sind diese vier Möglichkeiten äquivalent.

In [KM99] wurden diese Wahlmöglichkeiten beim Einfügen bereits thematisiert, wobei jedoch von maximal drei Möglichkeiten für die Einfügeposition ausgegangen wird (das Elternrecord, sowie ein linkes und ein rechtes Kinderrecord). In der genannten Quelle wird die bereits angesprochene Split Matrix dazu benutzt, die exakte physische Position für das Einfügen zu ermitteln. Auf den Fall, dass solche oder ähnliche Metadaten nicht zur Verfügung stehen, gehen die Autoren nicht ein. An dieser Stelle werden also neue Strategien benötigt.

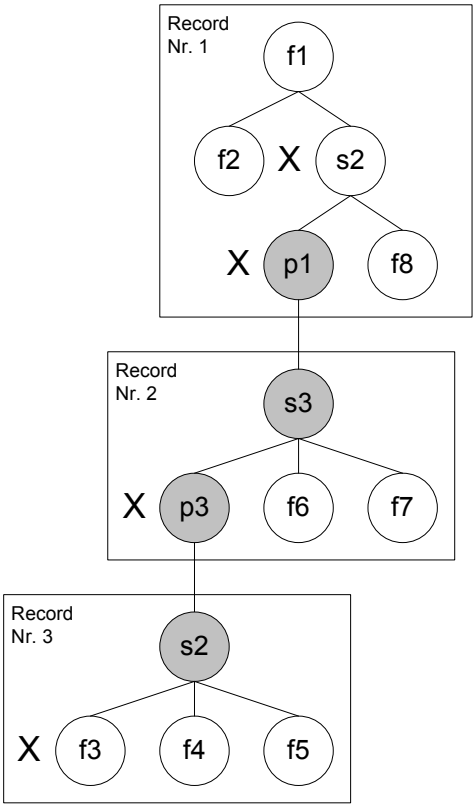


Abbildung 5.4: Wahlmöglichkeiten bei der exakten physischen Position einer Einfügeoperation. Es sind vier logisch äquivalente Positionen in der physischen Dokumentenstruktur markiert.

Hierfür schlage ich vier Heuristiken vor, die in Tabelle 5.1 aufgeführt sind. Wenn zwei äquivalente Positionen nach den Heuristiken gleich gut geeignet sind, so wird immer die erste gefundene Position verwendet (d. h. die verwendete Position ist in Dokumentenreihenfolge vor der zweiten gleichwertigen Position).

Name der Heuristik	Beschreibung
First (F)	Verwendet die erste gefundene Position und sucht auch nicht nach äquivalenten Positionen.
Smallest (S)	Durchsucht alle äquivalenten Einfügepositionen und nimmt diejenige, deren Record am kleinsten ist (die wenigsten Bytes belegt). Hierdurch soll erreicht werden, dass Splits möglichst lange verzögert werden.
Most below (MB)	Durchsucht alle äquivalenten Einfügepositionen und nimmt diejenige, die am weitesten unterhalb der ersten Position liegt, und zwar gemessen in der Länge des Pfades in der Anzahl der Knoten. Hierdurch soll erreicht werden, dass die oberen Ebenen der Speicherungsstruktur möglichst selten gesplittet werden müssen, da mit jedem Split die Qualität der Struktur potentiell sinkt.
Most proxies below (MPB)	Durchsucht alle äquivalenten Einfügepositionen und nimmt diejenige, die möglichst viele Records unterhalb der ersten Position liegt. In dem gefundenen Record wird dann die oberste, mögliche Position verwendet. Die Motivation für diese Heuristik ist ähnlich wie bei der letzten.

Tabelle 5.1: Heuristiken zur Bestimmung der exakten physischen Einfügeposition

5.4.2 Optimierung von Teilbäumen mit Artefakten

Durch Splits oder Löschoperationen können zwischenzeitlich Teilbäume entstehen, die eine Reihe von Artefakten besitzen. Beispielsweise sind dies Teilbäume mit den folgende Eigenschaften:

1. Der Teilbaum besteht aus genau einem Proxyknoten.
2. Es gibt einen Scaffoldknoten mit nur einem Kindknoten.
3. Es gibt einen Scaffoldknoten mit einem oder mehreren Scaffoldknoten als Kindknoten.
4. Es gibt einen Markupknoten mit einem Scaffoldknoten als Kindknoten.

Im ersten Fall sollte der Teilbaum immer sofort gelöscht werden. Der Proxyknoten des Teilbaums ersetzt hierbei in seinem Elternrecord den dort vorhandenen, gleich langen Proxyknoten, der den Teilbaum referenziert. Dieser erste Fall tritt nur dann auf, wenn viele Löschoperationen in einem Teilbaum durchgeführt wurden.

Fall zwei tritt ebenfalls bei Löschoperationen auf, wenn der zweite und letzte Teilbaum eines Scaffoldknotens entfernt wird. Die Auflösung des Artefakts ist hierbei einfach, da der Scaffoldknoten einfach durch seinen Kindknoten ersetzt werden kann.

Die Fälle drei und vier entstehen bei Natix-Splits, wenn der Separatorknoten als Wurzel einen Scaffoldknoten s besitzt und dieser in seinen Elternknoten unterhalb von einem Markup- oder Scaffoldknoten eingefügt werden muss. Hier kann s durch seine Kinderknoten im Baum ersetzt werden, wodurch Speicherplatz eingespart wird.

Alle diese Operationen sind relativ einfach und lokal durchführbar. Sie verbessern die Qualität des Baums erheblich, da die beschriebenen Artefakte bei den darauf folgenden Splits die Baumqualität weiter verschlechtern würden.

5.4.3 Splitalgorithmen

Ein Splitalgorithmus muss allgemein betrachtet einen Teilbaum, der zu groß für den Recordmanager ist, in mehrere Teile aufteilen. Für jeden Teil muss anschließend die Recorddarstellung kleiner als die maximale Recordgröße sein.

Ein Splitalgorithmus beschreibt seinen Split eindeutig durch eine Menge von Splitmarkierungen, die er im Baum R an die Knoten verteilt. Jede Markierung bedeutet, dass oberhalb des markierten Knotens k ein Split erfolgt und k der Wurzelknoten eines neuen Teilbaums wird. Splitmarkierungen können an jedem Knoten, außer an der Wurzel und außer an den Proxyknoten erfolgen. Somit gibt es für einen 1-fach Split, bei dem nur ein Teilbaum abgespalten wird, $|R|-1-|R.proxies|$ Möglichkeiten. Hierbei ist $R.proxies$ die Menge der Proxyknoten, die im Baum R vorhanden sind.

In diesem Sinne setzt der Natix Splitalgorithmus gleich mehrere Splitmarkierungen. Eine Markierung wird beim Splitknoten d gesetzt, die anderen am Rande des Separatorteilbaums.

In der Implementierung in XXL wird in jedem Knoten als Zusatzinformation die Größe des mit ihm beginnenden Teilbaums in Bytes gespeichert. Die Berechnung erfolgt zum einen beim Einlesen des Teilbaums durch den Converter, was sehr effizient möglich ist. Zum anderen müssen bei Operationen in Teilbäumen die geänderten Größen von der Stelle einer Änderung aus hoch propagiert werden. Diese Operation hat maximal einen Aufwand proportional in der Höhe des Teilbaums.

Für Splitalgorithmen bedeutet dies, dass der Zugriff auf Größeninformationen von Teilbäumen immer mit konstantem Aufwand erfolgt. Der Aufwand zur Aktualisierung der Größeninformationen nach einem Split geht i. d. R. im Aufwand des Splits selbst unter. In den folgenden Abschnitten wird auf die Aktualisierung der Größeninformation nach Änderungen in Teilbäumen nur bei Aufwandsabschätzungen eingegangen, nicht aber in den Algorithmen (aus Gründen der Übersichtlichkeit).

Nach der Implementierung des Separator Splits im XXL Gerüst zeigte sich, dass dieser Algorithmus in der Originalform aus der Veröffentlichung bei einigen Dokumenten nicht korrekt funktionierte. Wenn ein sehr großer Textknoten in einen Teilbaum eingefügt wird, der in diesem Fall ausschließlich in einem eigenen Record platziert werden kann, so kann der Split keinen erlaubten Separatorknoten finden, weil immer entweder die linke oder die rechte Partition größer als die maximale Recordgröße ($maxRecordSize$) ist.

Dieses Problem und weitere existieren für die meisten Splitalgorithmen, die im Folgenden vorgestellt werden. Daher besteht Bedarf an einer grundsätzlichen Lösung, auf die dann alle anderen Splitalgorithmen zurückgreifen können. Zunächst wird der hierzu entwickelte Splitalgorithmus *Simple Split*, vorgestellt, der definitiv jeden Teilbaum korrekt splitten kann. Anschließend werden zwei weitere Splitalgorithmen mit ihren Varianten beschrieben.

Alle diese drei Verfahren haben gemeinsam, dass es keine Separatoren gibt, die im XML Baum nach oben wandern. Es wird also keine Höhenbalancierung betrieben. Hierdurch kann sich der Baum besser den tatsächlichen Daten anpassen.

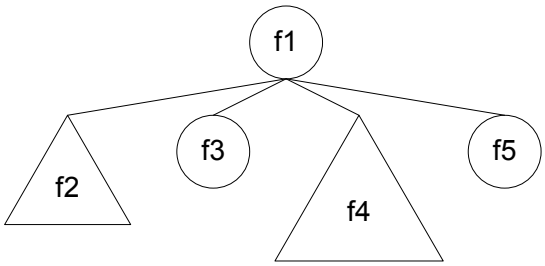
5.4.3.1 Der Simple Split

Der so genannte *Simple Split* ***SimS*** besucht in einer Tiefensuche alle Knoten des zu splittenden XML Baums. Bei dieser Suche wird in allen Knoten k angehalten, deren Teilbaum $T(k)$ gerade noch größer ist, als die maximal erlaubte Recordgröße. Das heißt, alle Teilbäume beginnend bei einem Kind von k besitzen eine kleinere Byterepräsentation als $maxRecordSize$.

An solchen Knoten erfolgt der eigentliche Split. Dieser läuft in zwei Phasen ab. In Phase 1 (siehe Abbildung 5.5) werden alle Teilbäume von $T(k)$, die eine vorgegebene Größe überschreiten, in eigene Records geschrieben. Ist danach immer noch $\|T(k)\| > maxRecordSize$, so werden weitere Teilbäume von $T(k)$ der Größe nach in eigene Records geschrieben, bis entweder $\|T(k)\| \leq maxRecordSize$ gilt oder $T(k)$ nur noch Proxyknoten als Kinder besitzt.

Bei Knoten mit hohem Verzweigungsgrad kann es vorkommen, dass auch nach dem Wegschreiben aller Kinderteilbäume der zu splittende Teilbaum immer noch größer als erlaubt ist. Nur in diesem Fall wird Phase 2 gestartet (siehe Abbildung 5.6). Der betrachtete Teilbaum muss entsprechend Abbildung 5.6.a aussehen. Zunächst müssen die Proxyknoten mittels neuer Scaffoldknoten gruppiert und in neue Records geschrieben werden. Dies erfolgt vom ersten Kindproxy ausgehend so lange, bis $\|T(k)\| < maxRecordSize$ gilt. Es kann passieren, dass diese Phase mehrfach durchlaufen werden muss.

a) vorher



b) nachher

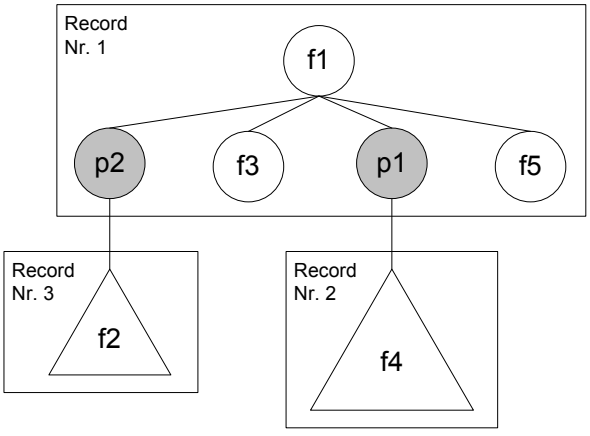
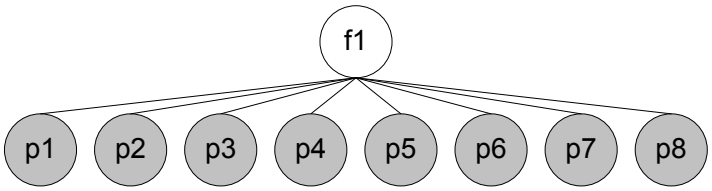


Abbildung 5.5: Phase 1 von Simple Split

a) vorher



b) nachher

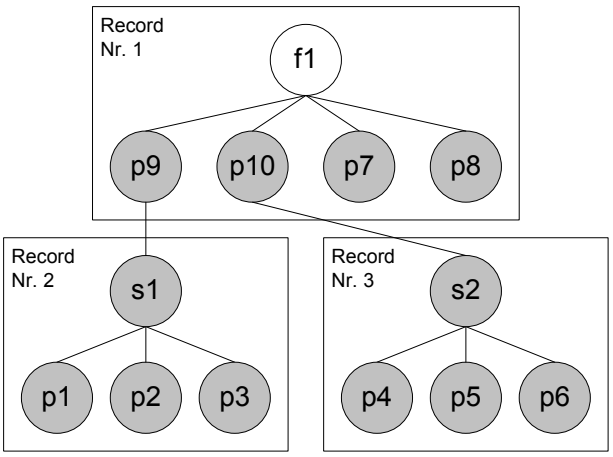


Abbildung 5.6: Phase 2 von Simple Split

Wenn der Split für einen Knoten erfolgt ist, so läuft die Tiefensuche im Baum weiter. Am Ende ist der komplette Baum so weit zerlegt, dass alle Teilbäume in Records Platz finden. Das komplette Verfahren ist noch einmal in Algorithmus 1 in Pseudocode aufgeführt.

Die CPU-Zeit von einem einzelnen Split (Phase 1 plus Phase 2 von einem Teilbaum ohne rekursiv auftretende Splits) ist proportional zur Anzahl der Knoten des gesplitteten Teilbaums. Das Hochpropagieren der neuen, veränderten Teilbaumgrößen kann generell in der Tiefensuche integriert erfolgen. Somit ist für die Anpassungen der Teilbaumgrößen nur eine lineare Laufzeit nötig. Hierdurch besitzt der Algorithmus für die Behandlung eines XML Baums R eine Laufzeit von $O(|R|)$.

Wichtiger als die CPU-Zeit ist jedoch die Anzahl der Externspeicherzugriffe. Hierüber kann leider nicht sehr viel ausgesagt werden. Es ist lediglich bekannt, dass für einen Split mindestens ein Zugriff und maximal $O(|R|)$ Zugriffe erfolgen müssen. Die obere Grenze ist allerdings wenig praxisrelevant, da sie sehr weit weg vom Normalfall liegt, bei dem mit einer konstanten Anzahl an erzeugten Records gerechnet werden kann.

5.4.3.2 Der Element Split

Der *Element Split* ***(ElS)*** wird bereits in einigen Systemen verwendet (TIMBER [JAKC+02], OrientStore [MLLA03]). Er ist ein relativ einfaches Verfahren und kann daher möglicherweise in Zukunft als Referenzverfahren dienen.

Ein Element Split sorgt dafür, dass jeder logische Knoten k des Baums (Markupknoten, Textknoten, Attribute etc.) in seinem eigenen Record gespeichert wird. Die Kindknoten werden über Proxyknoten referenziert. Eine zweite Variante dieses Splits, der *Element Split mit Attributen* ***(ElSA)***, versucht, die zu einem Knoten gehörenden Attribute im selben Record wie den Knoten selbst abzulegen. Dies bringt Vorteile, wenn Attribute sehr häufig angefragt werden.

Auch der Element Split muss in schwierigen Fällen auf den Simple Split ausweichen, beispielsweise dann, wenn der Verzweigungsgrad eines Knotens so groß ist, dass nicht alle Proxyknoten in ein Record passen.

Algorithmus 1 Der Simple Split

1: **procedure** SIMPLESPLIT(R,$threshold$)
2: Sei R der zu splittende XML Baum mit $\|R\|$>$maxRecordSize$.
3: Bemerkung: Beim Speichern eines Teilbaums werde dieser im folgenden immer automatisch durch einen Proxyknoten mit Verweis auf das erzeugte Record ersetzt.
4: **for** $k \in R.kinder$ **do**
5: **if** $\|T(k)\| > maxRecordSize$ **then**
6: SimpleSplit($T(k)$)
7: **end if**
8: **if** $\|T(k)\| > maxRecordSize$ **then** ▷ Phase 1
9: Sei $M = \{l | l \in T(k).kinder \wedge \|T(l)\| \geq threshold\}$
10: $\forall m \in M$: speichere $T(m)$ in einem Record
11: Sei $i = 1$
12: **while** $\|T(k)\| > maxRecordSize \wedge i \leq |T(k).kinder|$ **do**
13: Wenn $T(k).kinder[i]$ kein Proxyknoten ist, dann speichere $T(T(k).kinder[i])$.
14: i++
15: **end while**
16: **while** $\|T(k)\| > maxRecordSize$ **do** ▷ Phase 2
17: Sei n die maximale Anzahl von Proxyknoten, die unter einem Scaffoldknoten in ein Record passt.
18: Fasse Gruppen von n Kinderproxyknoten von $T(k)$ unter jeweils einem Scaffoldknoten s zusammen und speichere $T(s)$. Verfahre so, bis entweder alle Knoten behandelt wurden oder $\|T(k)\| \leq maxRecordSize$ gilt.
19: **end while**
20: **end if**
21: **end for**
22: **end procedure**

5.4.3.3 Der Onecut Split

Während der Separator Split auf logischer Ebene eine gute Aufteilung des XML Baums vornimmt, resultiert aus der physischen Aufteilung häufig eine Situation, in der sehr viele kleine Records entstehen. Dies führt bei den darauf folgenden Operationen zu sehr vielen Externspeicherzugriffen.

Der *Onecut Split* ***(OCS)*** versucht, auf der physischen Ebene eine bessere Aufteilung in zwei Teile zu erreichen. Dies geschieht, indem nur eine einzige Splitmarkierung verwendet wird. Diese Markierung soll an eine Stelle gesetzt werden, an der der darunterliegende Teilbaum möglichst genau $\|R\|/2$ Bytes belegt. Somit besitzen die beiden entstehenden Teile im besten Fall dieselbe Größe. Zum physischen Verbinden der beiden Teile wird nur ein einziger Proxyknoten benötigt, wodurch die Anzahl an virtuellen Knoten für einen Split minimal ist. Ein Beispiel für einen Onecut Split ist in Abbildung 5.7 zu finden.

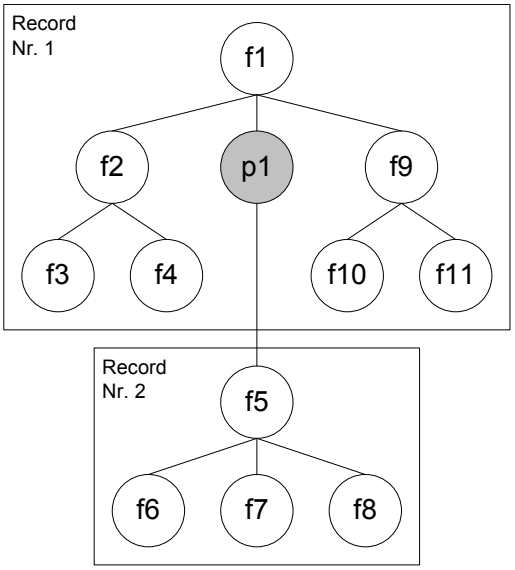


Abbildung 5.7: Aufteilung des Teilbaums aus Abbildung 5.3 mit einem Onecut Split

Der meiste CPU-Aufwand des Verfahrens liegt in der Suche nach einem geeigneten abzusplittenden Teilbaum. Das verwendete Verfahren zeigt Algorithmus 2. Die dort angegebene Funktion wird mit

$$best \texttt{ = FindOnecutSplitPosition(}R\texttt{,}R.wurzel\texttt{,}\|R\|/2\texttt{)}$$

aufgerufen. Wenn anschließend $best = R.wurzel$ ist, so ist kein adäquater Split möglich gewesen. In der Funktion `FindOnecutSplitPosition` wird erneut eine Tiefensuche durchgeführt, die jedoch abgebrochen wird, wenn in jedem Fall kein besser geeigneter Teilbaum unterhalb der aktuellen Position existieren kann (siehe die Bedingung in Zeile 9). Der Aufwand für den Onecut Split beträgt $O(|R|)$, da im Wesentlichen nur eine Tiefensuche durchgeführt wird.

Algorithmus 2 Das Auffinden einer Splitposition beim Onecut Split

```
1:  function FindOnecutSplitPosition(R, best, optSize)
2:      R ist ein Teilbaum des zu splittenden XML Baums.
3:      best ist der bislang beste Knoten für einen Onecut Split.
4:      optSize ist die gewünschte Größe des abzusplittenden Teilbaums.
5:      for e ∈ R.kinder do
6:          if |‖R‖ − optSize| < |‖T(best)‖ − optSize| then
7:              best = R.wurzel
8:          end if
9:          if ‖R‖ ≥ optSize then
10:             best = OnecutSplit(T(e), best, optSize)
11:         end if
12:     end for
13:     return best
14: end function
```

In der Praxis ist dieser Algorithmus jedoch häufig schlecht anwendbar, da in vielen XML Bäumen keine guten Aufteilungen gefunden werden können. Angenommen ein zu splittender Teilbaum besitzt 20 in etwa gleichgroße Kinderteilbäume. Dann ist der größte von ihnen ca. $maxRecordSize/20$ Bytes lang. Das Absplitten dieses größten Teilbaums führt dazu, dass der Ausgangsbaum nur minimal kleiner wird. Dementsprechend führt der Split nicht zu einem lang andauernden Erfolg. Schon die nächsten Änderungsoperationen auf dem Ausgangsbaum werden mit hoher Wahrscheinlichkeit einen weiteren Split notwendig machen. Dieser folgende Split ist dann wahrscheinlich ebenso schlecht. Die entstehende Baumstruktur bewirkt somit viele Externspeicherzugriffe bei den folgenden Operationen.

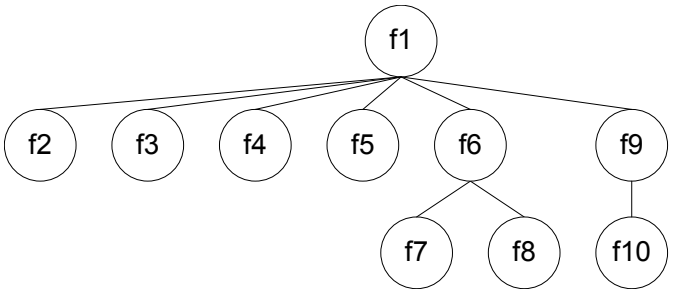
Dieser Fall von relativ flachen Strukturen kommt in der XML Realität sehr häufig vor, weshalb eine optimierte Variante des Splitalgorithmus, der *Onecut Split mit Scaffoldknoten* ***(OCSS)***, entwickelt wurde. Dieser Algorithmus versucht, eine Folge von benachbarten Knoten mit gemeinsamem Elternknoten zu finden, die serialisiert möglichst nah an die gewünschte, halbe Baumgröße herankommen. Wenn eine solche Folge von Knoten gefunden wird, dann werden diese an einen Scaffoldknoten gehängt und der entstandene Teilbaum in ein Record geschrieben. Dieses Verfahren benötigt also genau zwei physische Knoten für einen Split, einen Scaffoldknoten und einen auf ihn verweisenden Proxyknoten. Im Vergleich zum Separator Split ist dies immer noch ein sehr guter Wert. Ein Beispiel zeigt Abbildung 5.8.

Auf die ausführliche Darstellung des Algorithmus zum Auffinden solcher Folgen von Knoten wird an dieser Stelle verzichtet. Der Algorithmus ist sehr ähnlich zu Algorithmus 2, jedoch ist seine exakte Form in Pseudocode etwas länglich.

An dieser Stelle soll daher eine textuelle Beschreibung der Unterschiede im Vergleich zum vorangegangenen Algorithmus erfolgen. Es wird ebenfalls eine Tiefensuche durchgeführt. Auf jeder Ebene werden verschiedene Zusammenfassungen aufeinander folgender Kinderknoten betrachtet. Hierzu werden zwei Marker eingeführt, ein linker und ein rechter. Die dazwischen liegenden Knoten entsprechen einer möglichen Teilfolge für den abzusplittenden Baum. Zunächst werden beide Marker auf den ersten Kindknoten gesetzt. Ist der so gewonnene Teilbaum kleiner als $optSize$, wird der rechte Marker einen Knoten weiter gesetzt. Ist der Teilbaum größer als $optSize$, wird der linke Marker einen Knoten weiter gesetzt. Dies geschieht so lange, bis der rechte Marker beim letzten Kind angekommen ist und entweder der resultierende Teilbaum kleiner als $optSize$ ist oder der linke Marker ebenfalls beim letzten Kindknoten steht.

Für jede Teilfolge wird wie in Algorithmus 2 der Abstand zu $optSize$ berechnet. Die Berechnung der zu einer Teilfolge gehörenden Teilbaumgröße erfolgt inkrementell. Nach dem Versetzen eines Markers kann die neue Teilbaumgröße mit konstantem Aufwand berechnet werden. Der bisher beste gefundene Splitknoten wird in einer Variablen *best* gespeichert. Zusätzlich wird für die Vergleichsoperationen auch die zugehörige Teilbaumgröße in einer Variablen

a) vorher



b) nachher

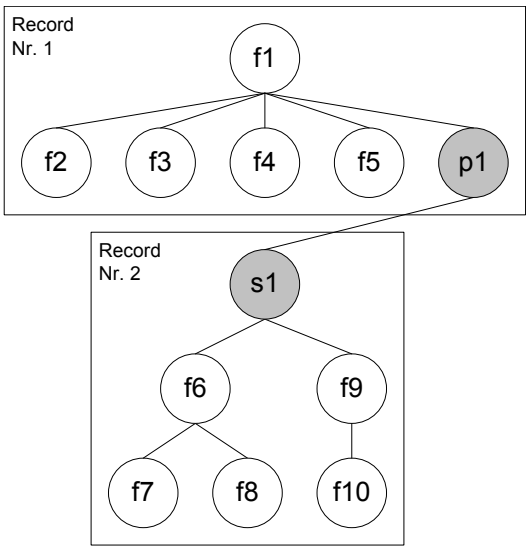


Abbildung 5.8: Aufteilung eines Teilbaums mit einem Onecut Split mit Scaffoldknoten

bestSize vermerkt. Ansonsten würde die wiederholte Berechnung der Größe aufeinanander folgender Knoten einen quadratischen Berechnungsaufwand bedeuten. Wie sich leicht erkennen lässt, liegt auch der CPU-Aufwand des OCS Algorithmus bei $O(|R|)$ Knotenbesuchen.

Beide Onecut Splitalgorithmen lassen sich noch durch die Einführung von Parametern steuern. Hierzu wird ein Intervall $I = [min, max]$ verwendet. Ein Split wird nur dann wirklich ausgeführt, wenn die Größe des abgesplitteten Teilbaums T im angegebenen Intervall I liegt, also $min \leq \|T\| \leq max$ gilt. Kann kein entsprechender Teilbaum gefunden werden, so tritt der Simple Split in Aktion.

Eine weitere Variante der Onecut Splitalgorithmen, der *Onecut Hybrid Split* ***(OCHS)*** funktioniert so, dass zunächst mit dem einfachen Onecut nach einem guten Split gesucht wird. Wenn ein Split möglich ist, der die vorgegebene Intervallbedingung erfüllt, so wird dieser Split durchgeführt. Ansonsten wird ein Onecut Split mit Scaffold versucht. Scheitert dieser ebenfalls, so wird wie zuvor auf den Simple Split ausgewichen.

5.4.4 Bulkloading

Splitalgorithmen bilden die Basis für ein System, das Einfüge-, Änderungs- und Löschoperationen auf XML Dokumenten unterstützt. Allerdings ist das Einfügen eines ganzen Dokuments mit einzelnen Einfügeoperationen sehr teuer. Für diesen Fall wird unbedingt ein Bulkloading-Mechanismus benötigt, der ein komplettes Dokument ohne viel Hauptspeicherverbrauch in die gewünschte Struktur transformiert.

In [KYU01] wird ein Algorithmus vorgestellt, der sukzessive Teilbäume einer vorgegebenen, maximalen Größe in einem Dokument zusammenfasst. Allerdings besitzen in dem dort beschriebenen Modell alle Knoten die gleiche Größe, was bei uns nicht der Fall ist. Außerdem ist der Algorithmus nicht effizient auf den Externspeicher anwendbar, da er in mehreren Phasen abläuft. Hierdurch müsste das Dokument mehrmals gelesen werden und zusätzlich müsste immer für jeden Knoten gespeichert werden, ob er schon bearbeitet wurde oder nicht.

Auf der nativen Speicherungsstruktur von Natix wurde bereits ein Verfahren zum Bulkloading von Dokumenten vorgeschlagen [Kan03]. Dieses Verfahren zeichnet sich dadurch aus, dass es mit einem SAX Durchlauf durch die Quelldatei auskommt.

Bei jedem Betreten eines Knotens wird eine Hauptspeicherrepräsentation von ihm und seiner Attribute erzeugt und diese an seinen Elternknoten, wenn vorhanden, angehängt. Beim Verlassen eines Knotens k wird geprüft, ob der unter ihm liegende Teilbaum $T(k)$ größer als $maxRecordSize$ ist. Wenn dies der Fall ist, so müssen Teilbäume der Kinderknoten so lange auf Platte geschrieben werden, bis $\|T(k)\| \leq maxRecordSize$ gilt[2]. Natix fasst beim Wegschreiben der Kinderteilbäume auf Platte diese beginnend beim am weitesten rechts stehenden Kindknoten zusammen. Hierdurch erhöht sich die Wahrscheinlichkeit dafür, dass das erste Kind zusammen mit dem Elternknoten gespeichert wird. Wenn mehrere aufeinanderfolgende Kindknoten in ein Record passen, so werden diese an einen neuen Scaffoldknoten gehängt und in einem Record gespeichert. Anschließend werden im Hauptspeicher die Kindknoten durch einen auf ihr Record verweisenden Proxyknoten ersetzt.

Bei einem hohen Verzweigungsgrad des Dokuments kann es passieren, dass ähnlich wie beim Simple Split Proxyhierarchien entstehen. Das Bulkloading läuft so lange, bis das Ende des Dokuments erreicht ist. Ganz am Schluss muss dann noch der im Hauptspeicher verbliebene Restbaum in ein Record geschrieben werden.

Nicht vergessen werden darf, dass in jedem Teilbaum (außer dem Wurzelteilbaum) die Recordidentifikatoren auf die Elternteilbäume gesetzt werden müssen. In dem oben beschriebenen Algorithmus stehen die Recordidentifikatoren der Elternteilbäume erst dann fest, wenn die Records in den Recordmanager eingefügt wurden. Zu diesem Zeitpunkt wurden die Kinderteilbäume jedoch schon längst geschrieben. Somit müssen die Kinderteilbäume nach der Verarbeitung der Elternteilbäume noch einmal eingeladen und aktualisiert werden. Hierzu sind zusätzliche Externspeicherzugriffe notwendig, die allerdings durch eine Pufferung deutlich reduziert werden können.

[2]Die Kinderteilbäume von k wurden zu diesem Zeitpunkt bereits bearbeitet, so dass ihre serialisierte Größe kleiner oder gleich $maxRecordSize$ ist.

Im Folgenden wird eine Analyse durchgeführt, die den benötigten Hauptspeicherbedarf von Bulkloading und zweier Varianten ermittelt. Vereinfachend wird davon ausgegangen, dass die maximale Größe eines Knotens k inklusive seiner Attribute in jedem Dokument d kleiner als die maximale Recordgröße ist, also

$$\max_{k \in d}(size(k)) \leq maxRecordSize$$

gilt mit $size(k) = \|k\| + \|k.attribute\|$, wobei $\|k\|$ die serialisierte Größe des Knotens k und $\|k.attribute\|$ die serialisierte Größe seiner Attribute ist. Weiterhin wurde in der Analyse angenommen, dass ein Baum im Hauptspeicher dieselbe Größe besitzt wie auf dem Externspeicher. In der Praxis ist die Hauptspeicherdarstellung i. d. R. ein wenig größer.

Der benötigte Hauptspeicher H für das Bulkloading eines Dokuments d beträgt in Natix maximal:

$$H \leq \max_{k \in d}(size(k)) \cdot (h(d) + 1) \cdot max_{fan}(d) + maxRecordSize$$

Hierbei ist $max_{fan}(d)$ der maximale Verzweigungsgrad, der im Dokument d vorkommt. Dieser Ausdruck kann noch ein wenig präzisiert werden, wenn alle Pfade des Dokuments betrachtet werden:

$$H \leq \left(\max_{p \in pfade(d)} \sum_{k \in p} \left[size(k) + \sum_{l \in vGeschwister(k)} size(l) \right] \right) + maxRecordSize$$

Hierbei ist $vGeschwister(k)$ die Folge von Geschwisterknoten des Knotens k, die in Dokumentenreihenfolge vor k stehen. Der maximale Hauptspeicher wird also für Pfade benötigt, in denen auf jeder Ebene sehr viele Knoten enthalten sind und diese auch viele Attribute besitzen.

Bei der ersten Variante dieses Verfahrens, die auch in XXL verwendet wird, erfolgt die Zusammenfassung der Knoten im Baum von links nach rechts. Hierbei können Folgen von Knoten schon dann in Records gespeichert werden, wenn in einem Geschwisterknoten festgestellt wird, dass die vorherigen Geschwisterknoten zusammen mehr Platz als die maximale Recordgröße belegen. Somit gilt für den maximalen Hauptspeicheraufwand H^* hier:

$$H^* \leq \left(\max_{p \in pfade(d)} \sum_{k \in p} size(k) \right) + 2 \cdot maxRecordSize$$

Die Größe des benötigten Hauptspeichers ist bei dieser Variante also im Wesentlichen abhängig von der Baumhöhe, wohingegen beim Natix-Bulkloading auch der Verzweigungsgrad eine Rolle spielt. In vielen Dokumenten ist der maximale Verzweigungsgrad allerdings größer als die Höhe des Dokuments. In XXL wurde aus diesem Grund diese Variante implementiert.

Eine weitere Reduzierung des Hauptspeicherbedarfs könnte dadurch erreicht werden, dass ein Knoten erst beim „Verlassen"-Ereignis von SAX[3]im Hauptspeicher aufgebaut wird. Hierdurch würde $size(k)$ im oberen Ausdruck zu $\|k.attribute\|$. Auf die Speicherung der Attribute kann allerdings nicht verzichtet werden, da ein SAX-Parser die Attribute nur beim Erreichen eines Knotens übermittelt und nicht beim Verlassen. Die Attribute müssen also in jedem Fall beim Erreichen eines Knotens zwischengespeichert werden, um sie beim Verlassen verarbeiten zu können. Für die Speicherung der Attribute kann ein Stack verwendet werden. Um weiter Hauptspeicher einzusparen, könnte dieser Stack auch einigermaßen effizient auf dem Externspeicher verwaltet werden. Dadurch würde sich der Hauptspeicheraufwand auf $H^{**} = maxRecordSize + 2B$ erniedrigen. $2B$ wird hier deshalb gewählt, damit im Fall, wenn der Externspeicherstack in der Nähe eines Blockwechsels auf und abgebaut wird, keine unnötigen E/A-Zugriffe verursacht werden.

Ein Vergleich der in XXL implementierten Variante von Bulkloading mit Natix Bulkloading und der vorgeschlagenen hauptspeicherschonenden Variante steht noch aus. Im Rahmen dieser Arbeit wurden Speicherungsbäume, die mit Bulk-

[3]Hiermit ist das Auffinden der schließenden Knotenmarkierung durch den Parser gemeint.

loading aufgebaut wurden, zum Test der Anfrageleistung verwendet. Wie sich später noch zeigen wird, ist die Anfrageleistung bei mit Bulkloading aufgebauten Strukturen deutlich höher als bei Strukturen, die unter Verwendung der Splitalgorithmen aufgebaut wurden. Somit kann Bulkloading als Maßstab dafür dienen, ob Verbesserungen in Splitalgorithmen überhaupt möglich sind.

5.4.5 Anfragefunktionalität

Ein XML Speicher macht nur dann Sinn, wenn er Anfragen in bestimmten standardisierten Formen effizient unterstützt. Zum einen sind diverse Sprachen für Anfragen auf XML Dokumenten von Bedeutung, die bereits in Kapitel 2.4.2 vorgestellt wurden. XPath nimmt unter diesen Sprachen die wichtigste Rolle ein, da es die Basis für nahezu alle weiteren, mächtigeren XML Sprachen bietet. Zum anderen sind in der Praxis Anbindungen an Applikationen von enormer Wichtigkeit. DOM- und SAX-Anbindungen gehören zum Pflichtumfang von nativen XML Datenbanksystemen. Weitere Anbindungen wie beispielsweise an JDOM [JDO04] sind durchaus üblich.

Im Folgenden geht es um Details der Implementierung der wichtigsten Anfragesprachen im nativen XML Speicher von XXL.

5.4.5.1 XPath

Die Implementierung von XPath ist wie bereits angesprochen absolut grundlegend für ein natives XML Datenbanksystem. Für unsere Zwecke ist eine vollständige Implementierung des XPath 1.0 Standards nicht notwendig, da die Anfragen nur dem Test der XML Speicherungsstruktur dienen sollen. Somit wurde eine Teilmenge von XPath implementiert, die jedoch die nötige Funktionalität für vorhandene Benchmarks anbietet.

Der XPath Parser in der Klasse `xxl.core.storage.utilities.XPathLocation` ist zunächst optimiert für das schnelle Parsen von einfachen XPath Ausdrücken. Diese werden häufig für die Angabe exakter, eindeutiger Lokationen in Dokumenten verwendet, beispielsweise bei der Angabe einer Einfügeposition-

on. Als Prädikat ist hier nur die Angabe einer Knotennummer erlaubt. Für Ausdrücke mit komplizierteren Prädikaten werden Klassen benutzt, die mittels des Parsergenerators ANTLR [Par04] aus einer Grammatik erstellt wurden. Das Parsen und die Auswertung dieser Ausdrücke ist deutlich zeitaufwendiger als bei den einfachen XPath Ausdrücken.

5.4.5.2 SAX

In das Gerüst des nativen XML Speichers wurde die Möglichkeit integriert, Ereignisse für ein Dokument ab einer bestimmten Position im Dokument zu generieren und diese an ein XXL Prädikat zu senden. Das Prädikat kann im Gegensatz zu SAX auf die Ereignisse reagieren und bestimmen, welche Kinderknoten besucht werden sollen und welche nicht. Meldet es `true` zurück, werden die Kinderknoten des aktuellen Knotens durchlaufen. Meldet es `false` zurück, so werden die Kinderknoten übersprungen.

Diese Art des ereignisbasierten Parsens von XML ist für nativen Speicher sehr effizient, da im Regelfall nicht die Ereignisse für den kompletten betrachteten Teilbaum generiert werden müssen. Dies wäre bei Standard-SAX der Fall, weil ein SAX-Eventhandler keine Möglichkeit bietet, den Parser zu beeinflussen. In normalen SAX-Parsern wurde dies nicht vorgesehen, weil beim Parsen von XML Dokumenten, die im Dateisystem liegen, eine entsprechende Steuerung nicht vorteilhaft ausgenutzt werden kann. Es muss in jedem Fall die komplette Datei gelesen werden.

Prinzipiell ließe sich mit nur wenig Aufwand ein spezielles XXL Prädikat schreiben, das die erhaltenen Ereignisse in SAX Ereignisse transformiert und diese an einen SAX-Eventhandler weiterreicht. Diese Funktionalität wurde allerdings in der Praxis bislang noch nicht benötigt. So ein Prädikat wäre in jedem Fall nur aus Gründen der Kompatibilität interessant. Für effiziente Verfahren wird die Implementierung eines entsprechenden XXL Prädikats vorgezogen.

5.4.5.3 DOM

Im Vergleich zur Implementierung einer ereignisbasierten Schnittstelle wie SAX, ist die Anbindung von DOM-Applikationen an den XML Speicher deutlich aufwendiger. Hier gibt es außerdem verschiedene Varianten, die für verschiedene Zwecke optimal sind.

Die erste Variante schreibt Teildokumente in Dateien, bearbeitet diese mit einem herkömmlichen DOM-Parser und importiert sie anschließend, falls Änderungen erfolgt sind, wieder in die Datenbank. Dieses Vorgehen ist für kleine Dokumente eine gute Variante, für größere jedoch nicht performant.

Die zweite Variante baut den kompletten DOM-Baum im Hauptspeicher auf, übergibt ihn der Applikation und schreibt hinterher die veränderten Knoten in die Datenbank zurück. Hierzu ist die Implementierung der kompletten DOM-API nötig, da Änderungsoperationen auf den Knoten registriert werden müssen. Allerdings verschwendet auch diese Technik die speziellen Fähigkeiten des nativen XML Speichers. Beide bisher vorgestellten Möglichkeiten sind darüber hinaus für wirklich große Dokumente nicht verwendbar, weil die Dokumentgröße nur einen Bruchteil des freien Hauptspeichers betragen darf.

An dieser Stelle versprechen persistente DOM-Modelle eine Verbesserung [HMF99]. Sie halten nur Knoten im Hauptspeicher, die momentan in der Applikation verwendet werden. Die Navigationsoperationen der DOM-API werden hierbei direkt in Navigationsoperationen auf dem nativen Speicher umgesetzt.

In XXL wurde dieser dritte Weg eingeschlagen. In der Implementierung werden alle Teilbäume, auf denen operiert wird, im Hauptspeicher gehalten. Wenn alle DOM-Knoten, die auf einem Teilbaum operieren, von der Applikation freigegeben werden, entscheidet der Garbage Collector von Java, wann welcher Hauptspeicherbereich wieder freigegeben wird.

Die XXL DOM-Anbindung bietet derzeit nur einen Lesezugriff (der Schreibzugriff ist in Entwicklung). Über diesen Lesezugriff können bereits jetzt ganze Anfragesprachen an den nativen Speicher von XXL angekoppelt werden. Beispielsweise bietet das Jaxen-Projekt [Jax04b] eine komplette Implementierung

des XPath-Standards auf Basis von DOM. Als eine höhere XML Sprache ist der Einsatz von XOQL [Agu04] geplant, dessen Implementierung ebenfalls prinzipiell auf der DOM-API basiert. Allerdings verwendet XOQL nicht ein einzelnes XML Dokument, sondern gleich eine ganze Kollektion von Dokumenten als Datenbasis. Da DOM leider nur für einzelne Dateien definiert ist, muss auf XOQL-Seite noch ein spezieller Anschluss geschaffen werden. Dieser soll die Abbildung von Dokumenten einer Kollektion auf verschiedene Dokumente des nativen Speichers erlauben.

5.4.6 Architektur

Die komplette Architektur des nativen XML Speichers in XXL zeigt Abbildung 5.9. Die unteren Ebenen sind bereits aus Abbildung 3.19 bekannt. Auf den Containern basiert nun der eigentliche native XML Speicher. Er kann entweder einen Recordmanager oder einen blockspeichernden Container verwenden. Bei letzterem ist mit einer deutlich niedrigeren Speicherplatzausnutzung zu rechnen.

Auf den Methoden der XML Speicherungsstruktur setzt zum einen die selbst implementierte und performante XPath Anfragebearbeitung auf und zum anderen die DOM-Schnittstelle, sowie die bereits beschriebene zu SAX ähnliche Parservariante. Auf der DOM-Schnittstelle steht durch die Verwendung des Jaxen-Projekts eine vollständige XPath-Implementierung zur Verfügung. Eine Applikation kann dann diese drei verschiedenen Möglichkeiten für ihre Zwecke verwenden.

5.4.7 Theoretische Aussagen

Theoretische Aussagen über die Splitalgorithmen sind leider nur schwer möglich. Entsprechende Aussagen sind auch in der Literatur bislang nicht zu finden. Ein Problem hierbei ist die Datenabhängigkeit. Für jeden Splitalgorithmus können Dokumente entworfen werden, mit denen er gut funktioniert, jedoch existieren immer auch Dokumente, bei denen Probleme der Algorithmen deutlich werden.

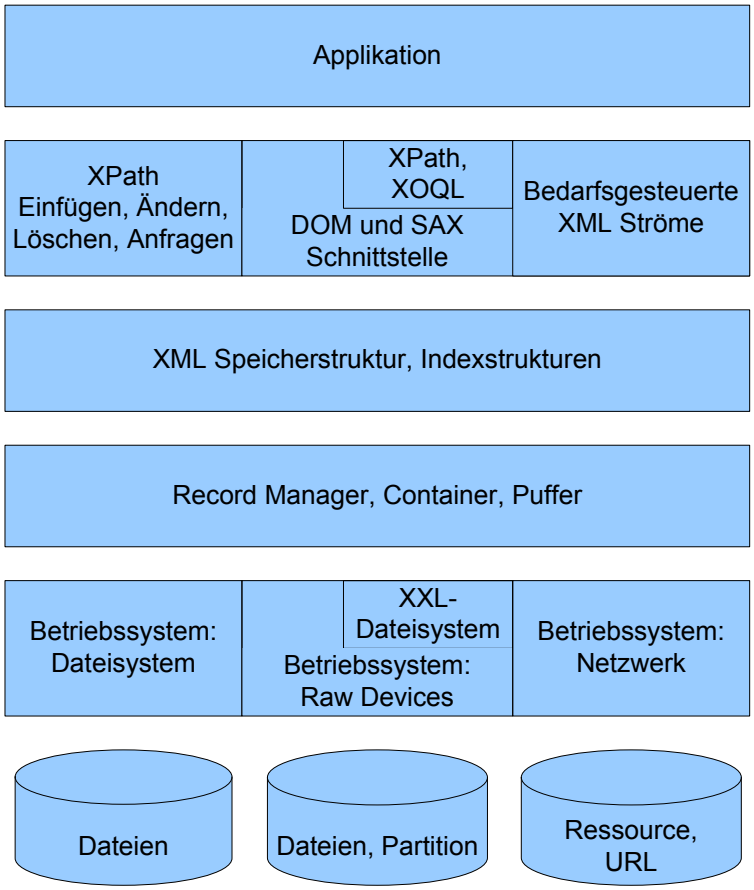


Abbildung 5.9: Die Ebenenarchitektur des nativen XML Speichers in XXL

Im Folgenden gehen wir von einer vereinfachenden Annahme aus: Ein Knoten k eines XML Dokuments passt immer zusammen mit allen seinen eventuell vorhandenen Proxyknoten in ein Record. Durch diese Einschränkung lassen sich einige wenige, allgemein gültige Aussagen treffen, die ansonsten nicht möglich gewesen wären. Die getroffene Annahme wird in realen Dokumenten in den allermeisten Knoten eingehalten und nur an Stellen mit sehr hohem Verzweigungsgrad lokal gebrochen. Die Annahme ist außerdem abhängig von der maximalen Recordgröße. Je größere Blöcke auf dem Externspeicher verwendet werden, desto realer ist die gemachte Annahme.

Beim Element Split muss unter dieser Voraussetzung in jedem Record genau ein realer Knoten liegen. Diese Art von Aussage ermöglicht Aufwandsabschätzungen für Navigationsoperationen. Die gewöhnlichen Navigationsoperationen (`getParent`, `getFirstChild`) erzeugen bei auf Element Split basierenden Strukturen also genau einen Externspeicherzugriff.

Beim Simple Split und bei der einfachen Variante des Onecut Splits gilt, dass immer **mindestens** die Wurzel des Teilbaums aus einem realem (=nicht virtuellen) Knoten bestehen muss. Es kann jedoch nicht ausgesagt werden, dass alle Pfade einer gewissen Länge $l > 1$ immer ausschließlich aus realen Knoten bestehen müssen. Eine solche Aussage würde den beiden Splitalgorithmen ein besseres theoretisches Verhalten als beim Element Split bescheinigen.

In allen drei, bisher beschriebenen Fällen ist die Höhe der Speicherungsstruktur (gemessen in der Anzahl an Records) durch die maximale Höhe des XML Dokuments nach oben beschränkt. Bei Dokumenten, die mit Bulkloading eingelesen wurden, gilt diese Eigenschaft übrigens ebenfalls.

Beim Onecut Split mit Scaffold können Records entstehen, deren Teilbäume ausschließlich aus virtuellen Knoten, also Proxy- und Scaffoldknoten, bestehen. Dies ist beispielsweise dann der Fall, wenn aus einem abgespaltenen Teilbaum (mit einem Scaffoldknoten als Wurzel) weitere Teilbäume abgespalten werden, und zwar so lange bis nur noch Proxies unterhalb des Scaffoldknotens übrig bleiben. Es kann also nicht garantiert werden, dass in jedem Record ein realer Knoten liegt und jede Navigationsoperation maximal zu einem Externspeicherzugriff führt. In der Praxis zeigt sich jedoch, dass der Onecut Split mit Scaffold

im Vergleich mit den anderen Splitalgorithmen häufig die Strukturen mit der geringsten Höhe produziert.

Beim Separator Split kann ebenfalls die Situation auftreten, dass sich ausschließlich virtuelle Knoten in einem Teilbaum befinden. Es kann sein, dass Separatoren nur aus einem Scaffold- und einem Proxyknoten bestehen. Wenn mehrere solche Separatoren zu einem Knoten hochpropagiert werden, kann der Fall eintreten, dass der Knoten danach nur aus virtuellen Knoten besteht.

Eine weitere theoretische Aussage zum Separator Split wurde bereits in Kapitel 5.3 getätigt. Da dieser Split beim Aufbau eines Dokuments einen höhenbalancierten Baum erzeugt, muss die Externspeicherdarstellung zwangsweise degenerieren, falls das Dokument selbst nicht ungefähr höhenbalanciert ist.

Eine wichtige Rolle spielt bei der Analyse der Speicherungsstruktur weiterhin die Anzahl an benötigten, virtuellen Knoten. Diese belegen Speicherplatz und machen bei übermäßigem Gebrauch die Struktur eventuell langsam. Zuerst sei bemerkt, dass bei größeren Blöcken auf dem Externspeicher die Anzahl an Splits deutlich zurück geht, und die Splitalgorithmen zusätzlich mehr Möglichkeiten für gute Splits erhalten. Weiterhin sind bei einer großer Blockgröße weniger virtuelle Knoten zur Verbindung der Teilbäume nötig.

Die meisten virtuellen Knoten werden vom Element Split produziert. Jeder Knoten außer der Wurzel besitzt einen auf ihn verweisenden Proxyknoten. Es existieren in einem Dokument mit n Knoten also immer $n - 1$ Proxyknoten. Allerdings werden wegen der oben gemachten Grundannahme überhaupt keine Scaffoldknoten benötigt.

Bei den anderen Splitalgorithmen entstehen im Vergleich zum Element Split weniger oder maximal gleich viele auf reale Knoten verweisende Proxyknoten. Jedoch können hier Proxyknoten hinzukommen, die auf virtuelle Knoten, also Scaffoldknoten, verweisen. Da diese jedoch normaler Weise in der Minderheit sind, sollten alle anderen Splitalgorithmen deutlich weniger Proxyknoten erzeugen, als der Element Split.

Ansonsten sind leider nur wenige Aussagen möglich. In Testläufen zeigt sich, dass durch einen Separator Split häufig sehr viele virtuelle Knoten erzeugt werden. Dies alleine lässt jedoch nicht auf eine hohe Anzahl an virtuellen Knoten

im Baum schließen, weil, wenn bei einem Split viele Records erzeugt werden, insgesamt die Anzahl an Splits niedriger sein kann als bei anderen Verfahren.

Virtuelle Knoten werden bei der Beseitigung von Artefakten (siehe Kapitel 5.4.2) zudem auch wieder gelöscht, beispielsweise, wenn beim Separator Split ein Separator mit einem Scaffoldknoten als Wurzel in seinen Vaterknoten unterhalb von einem normalen Knoten eingebaut wird. Dieses Vorgehen macht theoretische Vorhersagen über die Anzahl virtueller Knoten in der Speicherungsstruktur schwierig.

5.4.8 Praktische Evaluierung

In dem nun folgenden Abschnitt werden die verschiedenen Splitalgorithmen und das implementierte Bulkloading ausführlich getestet. Daten und Anfragen für die Experimente stammen vom XMark Benchmark [SWK+02], der noch genauer beschrieben wird. Zunächst erfolgt die Optimierung der noch festzusetzenden Parameter der Splitalgorithmen. Anschließend werden die besten Verfahren ausgesucht und einander gegenüber gestellt. Hierbei wird zwischen Einfüge- und Anfrageleistung unterschieden. Die Leistung beim Löschen von Knoten wird nicht weiter betrachtet, da sich die Verfahren hierbei ähnlich wie bei den Anfragen verhalten. Auch Einfüge- und Anfrageleistung sind übrigens nicht vollkommen unabhängig voneinander, da beim Einfügen eines Knotens immer zuerst eine einfache XPath-Anfrage gestellt werden muss, um die Einfügeposition zu lokalisieren.

Zum Abschluss der Evaluierung werden die besten Splitalgorithmen mit Bulkloading verglichen und einige Tests mit anderen Eingabedokumenten, sowie mit einem Objektpuffer präsentiert.

5.4.8.1 Der XMark Benchmark

Für die Evaluierung der implementierten Verfahren wurde der XMark Benchmark [SWK+02] eingesetzt. XMark simuliert das Szenario von einem Auktionshaus im Internet wie beispielsweise ebay. Er spezifiziert 20 Anfragen in der

Sprache XQuery. Die Anfragen werden an vordefinierte, verhältnismäßig große XML Dokumente gestellt. Diese müssen mit dem frei erhältlichen Werkzeug `xmlgen` generiert werden. Bei der Generierung kann ein Skalierungsfaktor angegeben werden, der die resultierende Dokumentgröße steuert und somit die absolute Laufzeit des Benchmarks beeinflusst. Die von `xmlgen` erzeugten Dokumente sind nicht übertrieben künstlich und kommen XML Dokumenten, wie sie in der Praxis zu finden sind, sehr nah.

Für unsere Zwecke wurden zwei relativ kleine Dokumente mit 120 KB und 1200 KB erzeugt. Da die Implementierung der Anfragesprache XQuery für die Tests der Splitalgorithmen als ein zu großer Aufwand erschien, wurden die in den 20 XQuery-Anfragen enthaltenen 33 XPath-Anfragen extrahiert (siehe Tabelle 5.2). In allen folgenden Tests werden dann nur noch diese XPath-Anfragen verwendet. In anderen Forschungsprojekten wurde ganz genauso verfahren [MLLA03].

Eine Schwäche des XMark Benchmarks ist das Fehlen von Änderungsoperationen. Die Autoren sind sich dieser Schwäche auch durchaus bewusst. Der Grund hierfür ist, dass zum Zeitpunkt der Entwicklung von XMark die Standardisierung in diesem Bereich noch nicht weit genug fortgeschritten war. Inzwischen wäre es möglich, unter Benutzung von XUpdate Änderungsoperationen für einen XML Benchmark zu definieren.

Da XMark sehr verbreitet ist und auch kein vergleichbarer Benchmark Änderungsoperationen vorsieht, wurde er für die Tests der nativen XML Speicherungsstruktur ausgewählt und eingesetzt.

5.4.8.2 Szenarien, Parameter und Messgrößen

In unserem Szenario arbeitet der native XML Speicher innerhalb eines XML Datenbanksystems auf einem Server. Er erhält Einfügeoperationen von verschiedenen Applikationen. Bei dieser Anfragelast werden Splitalgorithmen ganz besonders gefordert. Um dies in Tests zu simulieren, werden die einzelnen Knoten eines Eingabedokuments nach und nach in den nativen Speicher eingefügt. Hierbei muss die Nebenbedingung eingehalten werden, dass ein Knoten erst dann eingefügt werden kann, wenn sein Vaterknoten bereits existiert.

XPath Anfragen

```
01: /site/people/person[@id='person0']
02: /site/people/person[@id='person0']/name/text()
03: /site/people/person
04: /site/people/person/@id
05: /site/people/person/name/text()
06: /site/people/person/profile/interest/@category
07: /site/people/person/homepage/text()
08: /site/people/person/profile[@income>5000]
09: /site/people/person/profile[@income>=100000]
10: /site/people/person/profile[@income<100000 and @income>=30000]
11: /site/people/person/profile[@income<30000]
12: /site/open_auctions/open_auction
13: /site/open_auctions/open_auction/bidder[2]/increase/text()
14: /site/open_auctions/open_auction/bidder[1]/increase/text()
15: /site/open_auctions/open_auction/bidder/personref[@person='person18829']
16: /site/open_auctions/open_auction/bidder/personref[@person='person23']
17: /site/open_auctions/open_auction/reserve/text()
18: /site/open_auctions/open_auction/initial
19: /site/closed_auctions/closed_auction
20: /site/closed_auctions/closed_auction/price/text()
21: /site/closed_auctions/closed_auction/buyer/@person
22: /site/closed_auctions/closed_auction/annotation/description/parlist/listitem/
parlist/listitem/text/bold/text()
23: /site/closed_auctions/closed_auction/seller/@person
24: /site/regions
25: /site/regions/australia/item
26: /site/regions/australia/item/name/text()
27: /site/regions//item
28: /site/regions//item/name/text()
29: /site
30: /site//description
31: /site//annotation
32: /site//email
33: /site//item
```

Tabelle 5.2: Verwendete XPath-Anfragen innerhalb des XMark Benchmarks

Die gewählte Einfügereihenfolge für Tests sieht nun so aus, dass die Knoten zunächst nach ihrer Ebene im XML Dokument sortiert werden. Die Reihenfolge der Knoten, die sich auf dem gleichen Level befinden, ist dann zufällig (in den Experimenten allerdings immer identisch). Hierdurch wird der Zugriff mehrerer Applikationen simuliert, die in unterschiedlichen Teilen des Dokuments arbeiten.

Die wichtigsten Parameter für die Tests sind in Tabelle 5.3 aufgeführt. Als Eingabedatei wurde zunächst eine etwa 120 KB große Datei verwendet, die von `xmlgen` erzeugt wurde. Diese Dateigröße ist für den Test der Splitalgorithmen bereits vollkommen ausreichend. Im Recordmanager wurde grundsätzlich mit dem Verweiskonzept gearbeitet. Der LRU-Blockpuffer verwendet die write-back Strategie. Die Messgrößen sind zum Teil schon bekannt:

- Anzahl wahlfreier Zugriffe inklusive der Anzahl von Mehrfachzugriffen (im Folgenden mit „Zugriffen“ bezeichnet). Sequentielle Zugriffe spielen bei der Beurteilung der Verfahren keine Rolle.
- Rechenzeit (gemessen in Sekunden)
- Speicherplatzausnutzung
- Laufzeit mit direktem Festplattenzugriff (gemessen in Sekunden)
- Anzahl der virtuellen Knoten (Dies ist ein Gütekriterium für die Speicherungsstruktur)

Die Anzahl der Externspeicherzugriffe wird immer getrennt für die Phase des Baumaufbaus und für die Anfragephase gemessen. Wie sich bei den Experimenten zeigte, ist sie die bestimmende Größe für die Effizienz der Verfahren, da die Szenarien allesamt externspeicherlastig sind.

Bei der Rechenzeit wird ebenso zwischen Aufbau- und Anfragephase unterschieden. In der Anfragephase werden die XPath-Anfragen des XMark Benchmarks zehn Mal in Folge ausgewertet. Hierdurch wirkt sich auch für die ersten Anfragen ab dem zweiten Benchmarkdurchlauf die Pufferung aus.

Parameter	Beschreibung
d	Eingabedokument: XMark1, XMark2, Sha. XMark1 besitzt eine Größe von 120 KB, XMark2 von 1200 KB. Sha (160 KB) ist eine XML Version von Shakespeares Stück „The Comedy of Errors“ [Bos99]. Wenn keine Aussage über das verwendete Dokument gemacht wird, so wird das Dokument XMark1 eingesetzt.
B	Die Größe der Blöcke in Bytes: 512, 2048, 8192
P	Größe des Blockpuffer in Bytes: 0, 16384 (Art des Puffers: LRU Puffer mit write-back Strategie)
$Split$	Splitalgorithmus: ElS, ElSA, SimS, SepS, OCS, OCSS, OCHS
$PSRM$	Platzierungsstrategie des Recordmanagers. Es wurden verwendetet: AO, HY(10,0.85), LRULF(10), BFENFH(10) (siehe Kapitel 3.2.3.2)
EP	Heuristik für die Ermittlung der exakten Einfügeposition (siehe Kapitel 5.4.1): F, S, MB, MPB

Tabelle 5.3: Die Parameter beim Test des nativen XML Speichers

5.4.8.3 Optimierung der Parameter

Als erstes wurden eine Reihe von Experimenten durchgeführt, welche die freien Parameter der Splitalgorithmen festlegen sollen.

5.4.8.3.1 Separator Split (SepS) Beim Separator Split gibt es zwei festzulegende Parameter. Der erste Parameter, der **Split Faktor** f_{split}, legt die gewünschte Größenaufteilung zwischen der linken und der rechten Partition fest. Es gilt:

$$\frac{\sum_{L' \in L} \|L'\|}{\sum_{R' \in R} \|R'\|} = f_{split}$$

Ist $f_{split} = 1$, so wird versucht, dass beide Partitionen die gleiche Größe in Bytes besitzen. Bei $f_{split} > 1$ ist die linke Partition im Schnitt größer als die rechte, ansonsten umgekehrt.

Der zweite Parameter, die **Split Toleranz** t, gibt an, wie hoch der Teilbaum unterhalb des Splitknotens maximal sein darf. Somit wird die Anzahl der bei einem Split entstehenden Teilbäume reduziert. Es wird allerdings nach wie vor die beste, gefundene Aufteilung in die beiden Partitionen L und R verwendet, auch wenn der Teilbaum des gefundenen Splitknotens eine geringere Höhe als t besitzt. Bei gleich guten Aufteilungen wird immer diejenige mit dem höheren Separatorteilbaum bevorzugt.

Im Originalpapier von Natix [KM00] wurde nicht genauer ausgeführt, ob sich die Messung der Größe der Partitionen nach dem tatsächlich belegten Speicherplatz in Bytes (wie oben beschrieben) oder nach der Anzahl der Knoten richten soll. Zu Testzwecken wurde eine Variante von SepS namens *SepSK* implementiert, die auf den Anzahlen der Knoten basiert. In den Tests zeigte sich, dass SepSK im Schnitt 25.2% mehr Zugriffe für den Baumaufbau benötigt als SepS. Es wurde weiterhin weder beim Baumaufbau, noch bei den Anfragen eine Konstellation gefunden, in der SepSK besser als SepS war. Somit wird auf die weitere Darstellung von SepSK verzichtet.

Die Messreihe zum Separator Split umfasst insgesamt 3840 Einzelexperimente. Bei jedem Einzelexperiment wird ein Baum durch Einfügen der einzelnen Knoten wie oben beschrieben aufgebaut. Anschließend werden die Ergebnisse der Anfragen des XMark-Benchmarks berechnet. Im Folgenden werden die wichtigsten Resultate der Messreihe dargestellt.

Die erste zu beantwortende Frage betraf die Wahl der Heuristik zur Ermittlung der genauen Einfügeposition EP. Die Zugriffe für den Aufbau des Dokuments mit den verschiedenen Heuristiken sind in Abbildung 5.10 in Abhängigkeit zur Blockgröße aufgetragen. Da hier nicht alle Parameter fest gewählt wurden, erfolgte jeweils eine Mittelwertbildung über mehrere Messungen. Bei jedem nicht festgelegten Parameter wurde anschließend jeweils überprüft, ob die Wahl dieses Parameters das Messergebnis deutlich beeinflussen kann. Dies war bei keinem der nicht festgelegten Parameter der Fall.

Es zeigt sich, dass die MBP-Heuristik um bis zu 7% weniger Zugriffe erzeugt, als wenn in jedem Fall die oberste, mögliche Einfügeposition gewählt wird. MB erweist sich nur als minimal schlechter im Vergleich zu MBP (die Kurven liegen

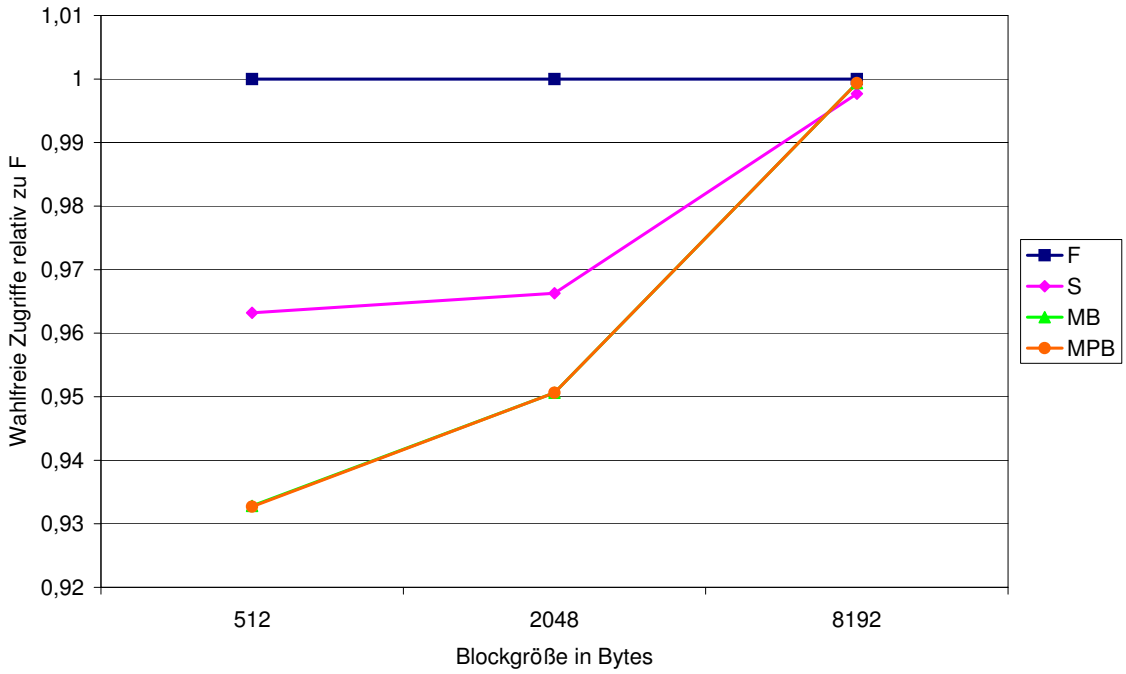


Abbildung 5.10: Relative Anzahl an Zugriffen beim Separator Split für unterschiedliche Blockgrößen und Heuristiken für die Einfügeposition

in Abbildung 5.10 praktisch aufeinander), wohingegen die Strategie S, die den Knoten in den Teilbaum mit dem meisten freien Speicherplatz einfügt, deutlich schlechter abschneidet. Ein weiteres Ergebnis ist, dass die Einfügestrategie immer unwichtiger wird, um so größer die verwendeten Blöcke sind. Bei 8KB großen Blöcken sind alle Varianten eng beisammen.

Der Einfluss der Einfügeheuristik auf die Leistung bei Anfragen ist nicht so groß wie der oben dargestellte Einfluss auf den Einfügevorgang. Generell ist jedoch das Ergebnis dasselbe, also die Rangfolge der Heuristiken identisch. Bei den weiteren, dargestellten Experimenten mit SepS wird nun ausschließlich die MBP Strategie verwendet.

Die nächste Frage betraf die Wahl der Splitparameter. In Abbildung 5.11 sind drei Diagramme zu finden, die die Anzahl der Zugriffe während der Aufbauphase enthalten. Die Zugriffszahlen sind hierbei farblich kenntlich gemacht[4].

[4]Die Diagramme sind eigentlich 3D-Diagramme, die von oben betrachtet werden. Der Bereich zwischen zwei Höhenlinien wurde eingefärbt.

Innerhalb der gewählten Grenzen ($0.7 \leq f_{split} \leq 1.3$ und $0 \leq t \leq 3$) liegen die Unterschiede in der Anzahl der Zugriffe bei maximal 30%. Die besten Werte ergeben sich bei $f_{split} = 1.15$ und $t = 2$. Diese Werte werden daher im Folgenden verwendet. Bei kleineren Seitengrößen kann insbesondere f_{split} eher etwas größer gewählt werden, jedoch sind auch in diesem Fall die festgelegten Werte immer nah an den besten Resultaten.

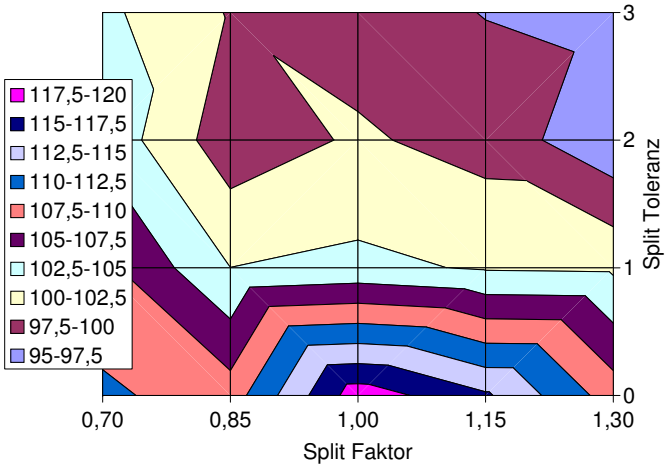
Bei der Wahl der genannten Splitparameter wurde zusätzlich darauf geachtet, dass die hier noch nicht beschriebenen Messgrößen gute bis sehr gute Werte liefern. Die Wahl der Splitparameter ist also für die Messgröße „Anzahl der Externspeicherzugriffe“ korrekt und weiterhin aus Sicht der anderen Messgrößen nicht falsch.

Theoretisch erklären lassen sich die Messergebnisse wie folgt. Ein größerer Split Faktor als eins ist deshalb sinnvoll, weil hierdurch die Position des Splitknotens weiter nach rechts im Teilbaum verschoben wird. Somit bleiben mehr von den linken, häufiger benötigten Kinderknoten nach dem Split in einem Record zusammen. Ein höheres t führt dazu, dass insgesamt weniger Records erzeugt werden, was im Schnitt größere Teilbäume und somit weniger Zugriffe zur Folge hat.

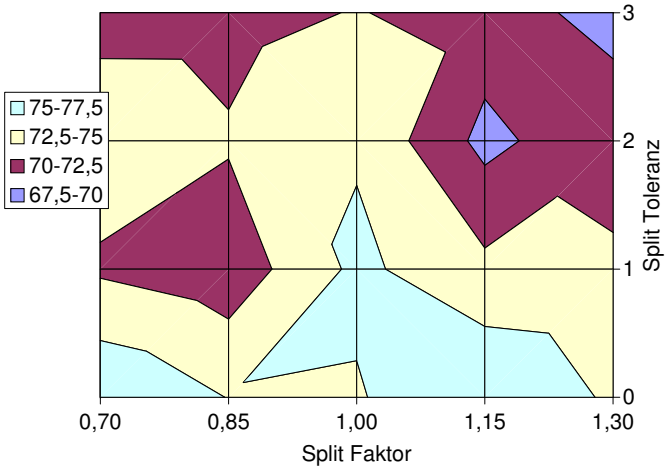
Als letzte Messgröße wollen wir an dieser Stelle noch den Rechenaufwand betrachten. Als Parameter sind hierbei verschiedene Blockgrößen und die Strategien des Recordmanagers interessant (siehe Abbildung 5.12[5]). Es ist zu erkennen, dass die Platzierungsstrategie des Recordmanagers praktisch keinen Einfluss auf den Rechenaufwand besitzt. Dafür wirkt sich die Blockgröße sehr deutlich aus. Die Ursache hierfür ist der Baumkonverter. Er konvertiert immer komplette Teilbäume, von denen häufig nur kleine Bereiche für Operationen benötigt werden. Je größer die Blockgröße ist, desto größer sind diejenigen Bereiche der Teilbäume, die gar nicht durchlaufen werden. Es besteht hier in etwa ein linearer Zusammenhang. Somit steigt auch die Rechenzeit linear mit B an.

[5] In der Abbildung wurden die Messpunkte mit gleicher Platzierungsstrategie des Recordmanagers wegen der besseren optischen Wahrnehmung miteinander verbunden.

a) B=512



b) B=2048



c) B=8192

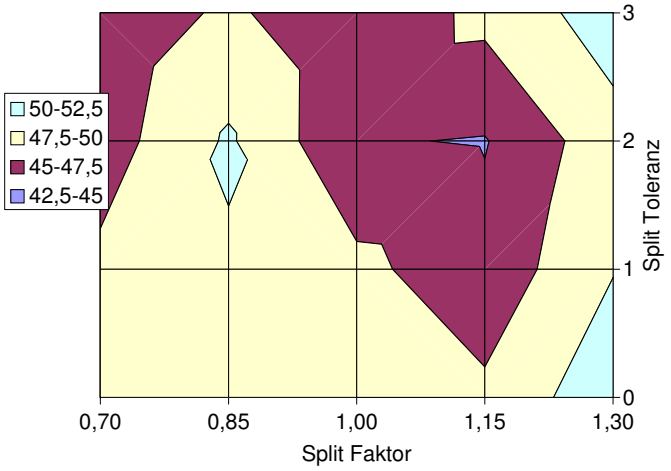


Abbildung 5.11: Anzahl der Zugriffe für den Aufbau eines Dokuments mit dem Separator Split bei verschiedenen Splitparametern und Blockgrößen. EP=MPB

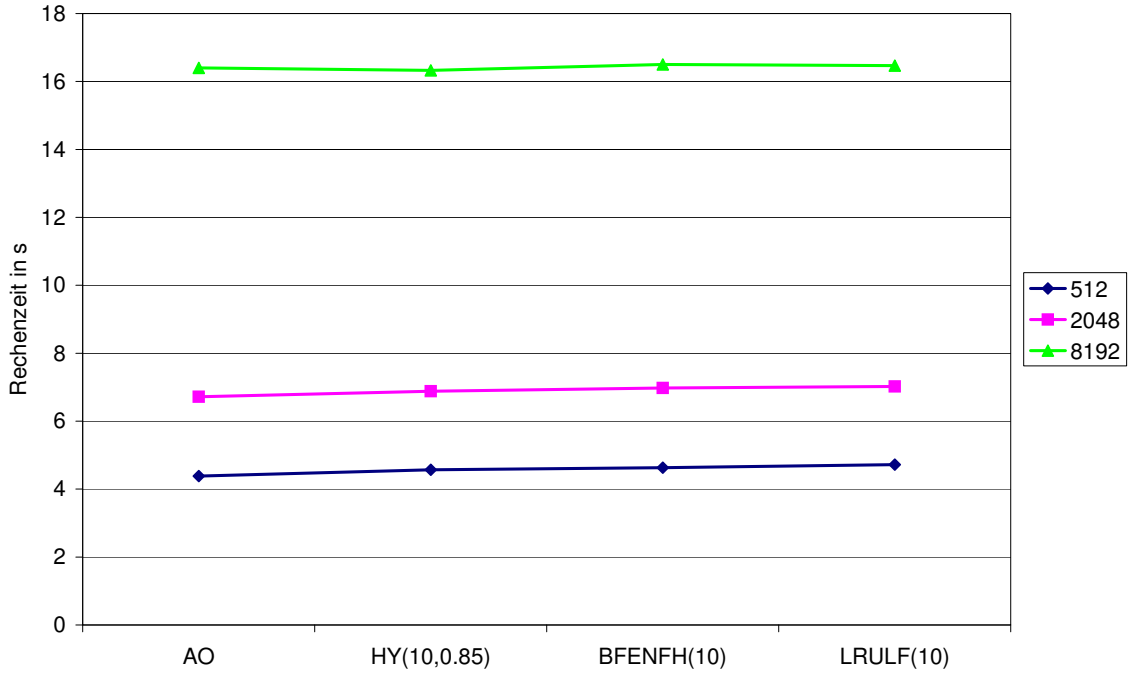


Abbildung 5.12: Rechenaufwand für den Separator Split für unterschiedliche Blockgrößen und Strategien des Recordmanagers. EP=MPB, $f_{split} = 1.15$, $tolerance = 2$

5.4.8.3.2 Onecut Split (OCS) Bei den Onecut Splits wurde zunächst ebenfalls die Auswirkung der Einfügeposition analysiert. Hier zeigt sich ein anderes Bild als beim Separator Split. Bei allen drei Varianten vom Onecut Split sind die Heuristiken in etwa gleich gut. Es gibt nur kleine Abweichungen nach oben und unten. Somit ist die Wahl der Heuristik für Onecut Splits nicht so wichtig. MPB zeigt insgesamt minimal bessere Ergebnisse, weshalb sie im Folgenden bei Onecut Splits verwendet wird.

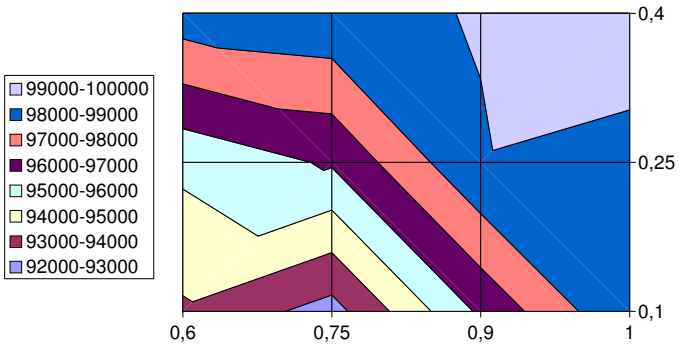
Als nächstes sollen auch für die Onecut Splits die optimalen Split-Parameter bestimmt werden. Zur Erinnerung: Die Verfahren werden mit einem Intervall I parametrisiert. Wenn die Größe des abzuspaltenden Teilbaums geteilt durch die Blockgröße im Intervall I liegt, so wird der Split ausgeführt. Wenn nicht, dann wird auf einen anderen Split umgeschaltet (Simple Split bzw. Onecut Split ohne Scaffold).

Zunächst zum Onecut Split mit Scaffold. Hier wirken sich die Parameter fast gar nicht aus. Das liegt daran, dass fast immer ein Split gefunden werden

kann, der sehr nah an die gewünschte, optimale Teilbaumgröße herankommt. Die zweite Beobachtung ist, dass die Wahl der Parameter unwichtiger wird, je größer B ist. Bei $B = 8192$ haben Parameter außerhalb des Intervalls $I = [0.4, 0.6]$ überhaupt keine Auswirkung mehr, da alle abgespaltenen Teilbäume die Intervallbedingung sowieso erfüllen.

Beim OCS sieht die Situation völlig anders aus (siehe Abbildung 5.13.a). Das Intervall $I = [0.1, 0.75]$ zeigt hier die besten Ergebnisse, und zwar weitgehend unabhängig von der Blockgröße. Es ist also wichtig, den Onecut Split auch dann auszuführen, selbst wenn die Größe des abzuspaltenden Teilbaums bei nur 10% liegt. Der Simple Split führt in diesem Fall offensichtlich zu schlechteren Ergebnissen und muss daher vermieden werden.

a) OCS



b) OCHS

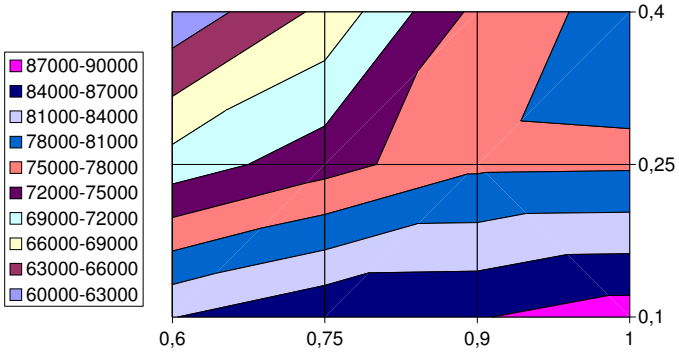


Abbildung 5.13: Anzahl der Zugriffe für den Aufbau eines Dokuments mit Onecut Split (OCS) und Onecut Hybrid Split (OCHS) bei verschiedenen Splitparametern. x-Achse: max(I), y-Achse: min(I), EP=MPB

Beim Onecut Hybrid Split (OCHS) sieht die Situation eine Ebene höher ähnlich aus (Abbildung 5.13.b). Hier muss der einfache Onecut Split vermieden werden,

da der Onecut Split mit Scaffold immer zu besseren Bäumen führt. Somit muss das Intervall I möglichst klein gewählt werden. Von den Testparametern eignete sich $I = [0.4, 0.6]$ am Besten. Wenn das Intervall noch kleiner gewählt wird, so tritt der Effekt auf, dass der Onecut Split mit Scaffold ebenfalls scheitert und der Simple Split genommen werden muss. In diesem Fall böte sich eventuell die Parametrisierung des OCHS mit zwei Intervallen an. Dies wurde jedoch nicht weiter untersucht.

Beim Rechenaufwand zeigt sich ein ähnliches Bild wie beim Separator Split. Eine größere Blockgröße B macht sich deutlich negativ bemerkbar, wohingegen die Strategie des Recordmanagers keine Auswirkungen hat.

5.4.8.3.3 Element Split Bei den beiden Varianten des Element Splits gibt es nur einen Parameter, den es festzulegen gilt, und zwar die Wahl der Heuristik für die Einfügeposition. Zunächst eine Vorüberlegung: In einem mit einem Element Split aufgebauten Dokument gibt es nur dann überhaupt Scaffoldknoten, wenn der Simple Split wegen eines hohen Verzweigungsgrads benutzt werden musste. Dies ist nur bei kleinen Blockgrößen der Fall und auch dort relativ selten. Ansonsten existiert im Baum kein einziger Scaffoldknoten und daher auch keine Wahlmöglichkeit bei der Einfügeposition. Das Suchen der Einfügeposition erzeugt somit unnötige Externspeicherzugriffe. Daher ist bei diesen Splitalgorithmen grundsätzlich immer die erste Einfügeposition zu wählen ($EP = F$).

5.4.8.3.4 Simple Split Beim Simple Split liegt die erste Einfügeposition praktisch gleichauf mit der Heuristik S. Die anderen Strategien folgen knapp dahinter mit einem guten halben Prozent mehr Zugriffe. Da die erste Einfügeposition auch immer von der Rechenlast her die Beste ist, wird beim Simple Split die Heuristik F verwendet.

5.4.8.4 Gegenüberstellung der Splitalgorithmen

Den Vergleich zwischen den so optimierten Splits zeigt Abbildung 5.14. Als Blockgröße wurde $B = 512$ Bytes festgelegt, da insbesondere die schnellen

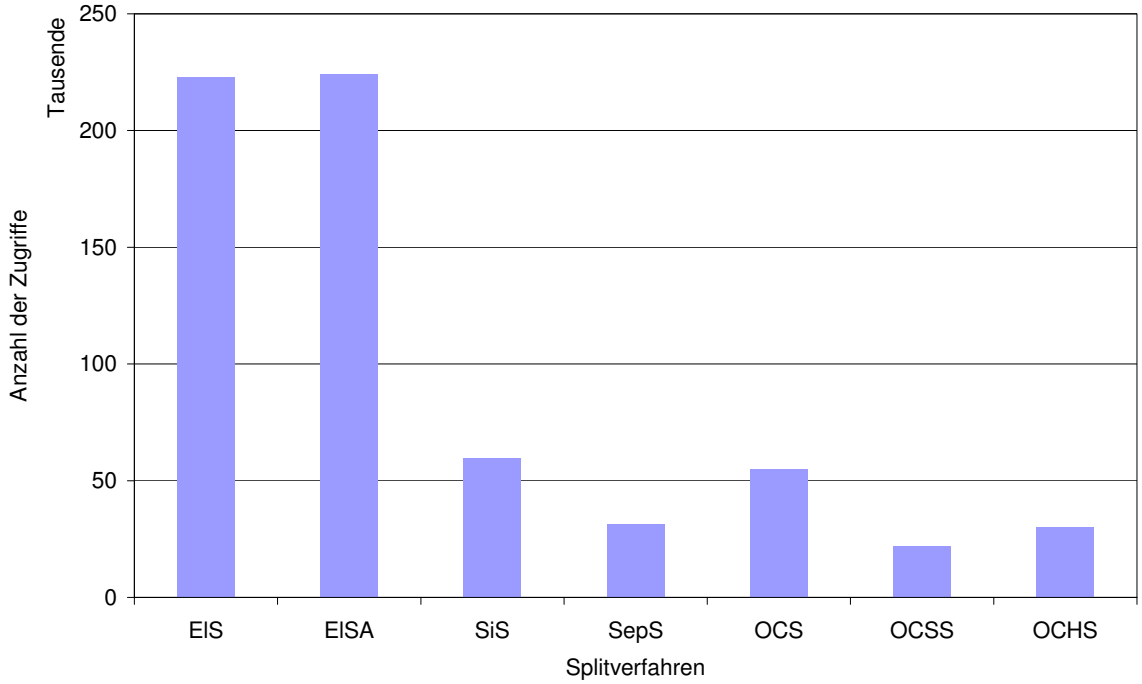


Abbildung 5.14: Anzahl der Zugriffe für den Baumaufbau mit den verschiedenen Splitalgorithmen. B=512, P=16384

Verfahren bei dieser Blockgröße die wenigsten Zugriffe benötigen. Zusätzlich gilt, dass Zugriffe auf größere Blöcke immer ein wenig teurer sind[6]. Der Puffer wurde auf 16KB gesetzt, weil dies unter den Bedingungen angemessen erschien.

Es ist zu sehen, dass der optimierte Natix Split im Vergleich zum neu entwickelten Onecut Split mit Scaffold im Schnitt 41.8% mehr Externspeicherzugriffe benötigt. Auch der Onecut Hybrid Split ist noch knapp schneller als der Separator Split.

Der Simple Split wird in jedem nativen XML Datenbanksystem des beschriebenen Typs unbedingt benötigt, weil er schwierige Fälle in XML Dokumenten behandelt. Bei den Messungen schneidet der Simple Split deutlich schlechter ab als das beste Verfahren (170% mehr Zugriffe als bei OCSS). Daher scheidet

[6]Bei der verwendeten Festplatte gibt es im Schnitt 319 Sektoren pro Spur. Somit dauert das reine Lesen von einem Sektor $t = \frac{1}{319} \cdot \frac{60}{7200} s$ (nur der mechanische Zeitanteil). Das Lesen eines Blocks von 16 Sektoren dauert dementsprechend 16 Mal so lange. Wenn das Erreichen des ersten Sektors bei einem wahlfreien Zugriff in etwa die Zeit für eine Plattenumdrehung benötigt, so dauert der Zugriff auf 16 Sektoren etwa 4.7% länger als der Zugriff auf einen einzelnen Sektor. Je schneller die Suchoperation eines Zugriffs ist, desto größer ist der relative Zeitvorteil von kleinen Blöcken.

die Option aus, in einem realen System ausschließlich diese Splitart anzubieten. Aus Leistungsgründen wird immer ein zusätzlicher Splitalgorithmus benötigt.

Der einfache Onecut Split ist in der Regel etwas besser als der Simple Split. Dies war erwartet worden, weil der Onecut Split stark auf dem Simple Split aufbaut und diesen partiell verbessert. In den Fällen, in denen OCS einen relativ großen Teilbaum mit nur einem Proxyknoten abspalten kann, wird nicht der Simple Split, sondern diese bessere Splitmöglichkeit verwendet. Da dies in der Praxis nur bei etwa 10% der zu splittenden Teilbäume gut funktioniert, liegen die Leistungen von SiS und OCS nicht weit auseinander.

Die Element Splits schneiden in den Experimenten deutlich schlechter ab. Sie sind die Schlusslichter und dienen nur als Referenzverfahren.

Bei der Anfrageleistung ist die Situation ähnlich (siehe Abbildung 5.15). Es gibt zwei erkennbare Unterschiede zu den vorherigen Experimenten. Erstens steigt der Vorteil von OCSS im Vergleich zu SepS von 41.8% auf 55.6% an. Zweitens liegt ElS deutlich vor ElSA, was daran liegt, dass bei den Anfragen von XMark nur vergleichsweise wenige Attribute angefragt werden und daher bei ElS weniger Teile des Dokuments betrachtet werden müssen. Somit reduziert sich für ElS die effektive Datenmenge. Im Folgenden wird auf die Darstellung der Messergebnisse des ElSA-Verfahrens vollkommen verzichtet, da es im Zweifelsfall immer schlechter abschneidet als ElS.

Die Einfügeleistung von Bulkloading ist um vieles besser als die Einfügeleistung des besten Splitalgorithmus. In den Experimenten zeigt sich, dass bei $B = 512$ der Onecut mit Scaffold etwa 25 Mal mehr Zugriffe benötigt als Bulkloading (ohne Grafik). Mit steigender Blockgröße B erhöht sich der Vorteil von Bulkloading im Vergleich zu den Splitalgorithmen noch weiter, da dann deutlich weniger wahlfreie Zugriffe für die Aktualisierung der Aufwärtsverknüpfung im Baum nötig sind.

Interessant ist, dass sich Bulkloading zusätzlich sehr positiv auf die Anfrageperformanz auswirkt. In Abbildung 5.16 ist die Anzahl der Zugriffe auf den Externspeicher für die Splitalgorithmen[7] und für Bulkloading aufgetragen. Es

[7] Zur besseren Vergleichbarkeit wurden die Element Splits hier weggelasssen. Die Zugriffszahlen für die

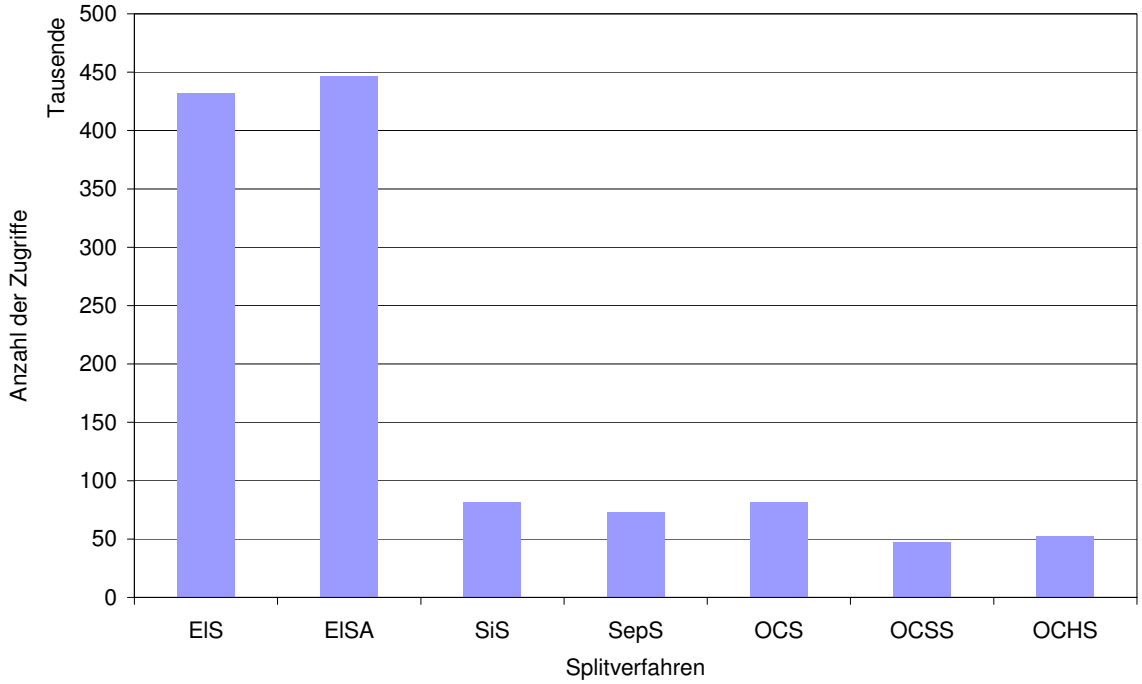


Abbildung 5.15: Anzahl der Zugriffe für die XMark Anfragen auf Bäume, die mit den verschiedenen Splitalgorithmen erstellt wurden. B=512, P=16384

zeigt sich, dass die Anfragen auf der durch Bulkloading erzeugten Struktur um einen Faktor von etwa 2.5 schneller sind, als beim besten Splitalgorithmus OCSS. Eine nennenswerte Abhängigkeit von der Blockgröße wurde hier nicht festgestellt.

Die Anfrageleistung hängt wesentlich von der Güte der erstellten Bäume ab. Ein wichtiges Gütekriterium ist die Anzahl von virtuellen Knoten. Dies ist in Abbildung 5.17 dargestellt. Die Anzahl an Proxyknoten p ist wichtig, weil sich aus ihr die Anzahl an Records r ableiten lässt: $r = p + 1$. Eine höhere Anzahl an Records führt zu mehr Sprüngen im Recordmanager und somit zu mehr Externspeicherzugriffen. Die Zahl an Scaffoldknoten wirkt sich dahingegen nicht so offensichtlich auf die Gesamtleistung aus.

Weitere Gütekriterien sind die Größe des durch den `BlockFileContainer` belegten Speicherplatzes auf dem Externspeicher (im Folgenden als Problemgröße bezeichnet), sowie die Speicherplatzausnutzung (SPAN). Diese beiden Größen

Splitalgorithmen entsprechen exakt denen aus Abbildung 5.15.

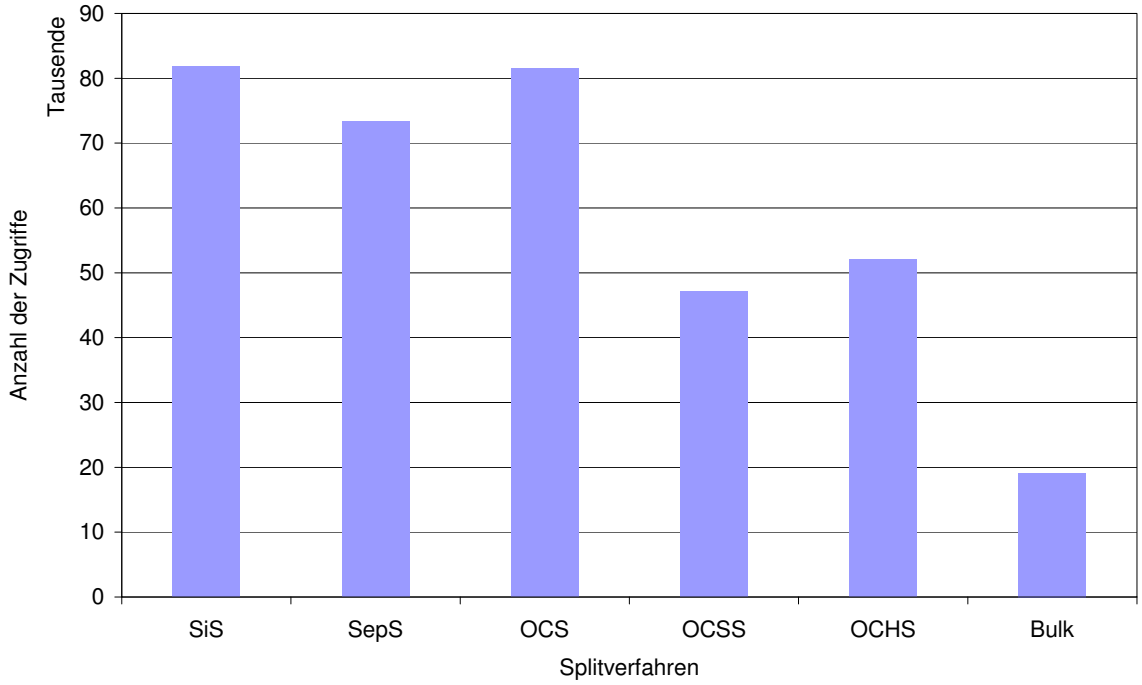


Abbildung 5.16: Anzahl der Zugriffe für die XMark Anfragen bei verschiedenen externen Baumstrukturen. B=512, P=16384

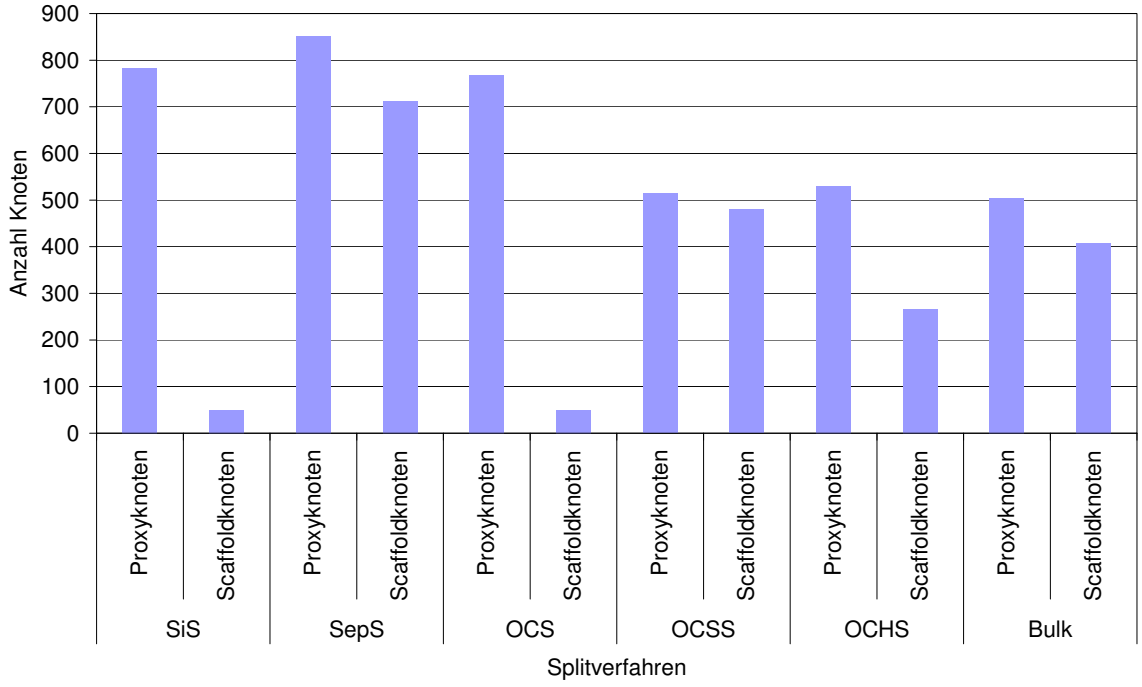


Abbildung 5.17: Anzahl von virtuellen Knoten bei den verschiedenen externen Baumstrukturen. B=512, P=16384

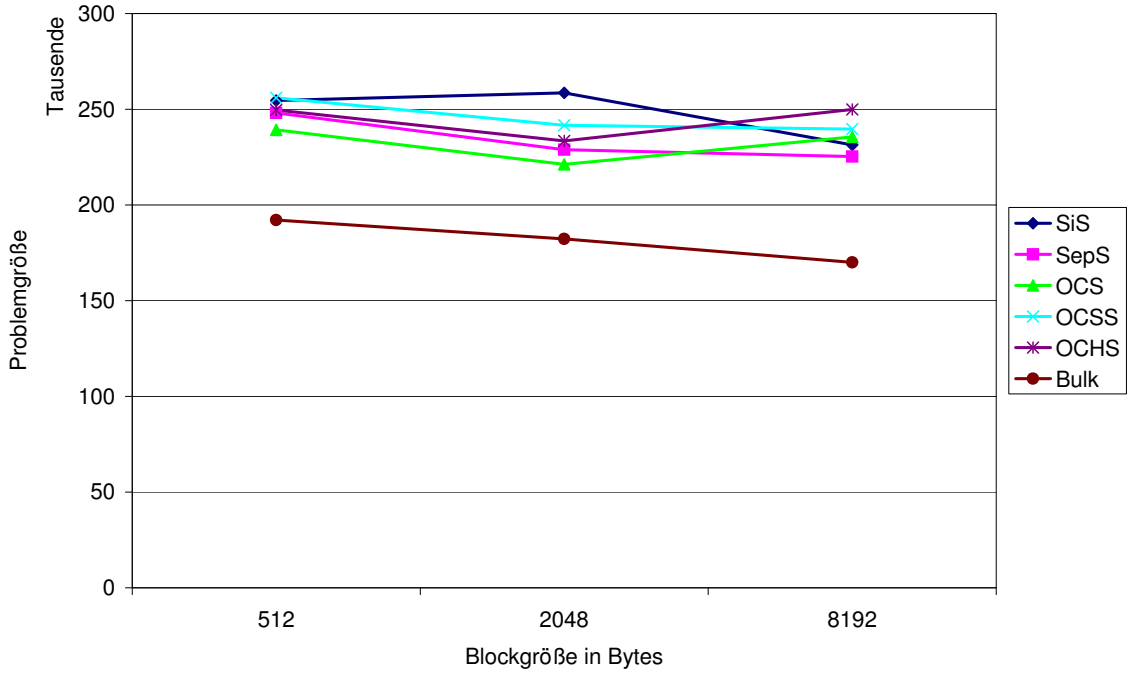


Abbildung 5.18: Größe des belegten Speicherplatzes der verschiedenen Verfahren in Abhängigkeit zur Blockgröße

müssen immer gemeinsam interpretiert werden. Verfahren, die viele virtuelle Knoten produzieren, erhöhen die Problemgröße und können trotzdem eine gute Speicherplatzausnutzung besitzen. In den Abbildungen 5.18 und 5.19 sind die beiden Messgrößen in Abhängigkeit von der Blockgröße und vom Verfahren aufgetragen.

Bei der Problemgröße setzt Bulkloading die Maßstäbe. Die Ursachen hierfür sind die sehr gute Speicherplatzausnutzung - die beste SPAN von allen Verfahren - und gleichzeitig eine niedrige Anzahl an virtuellen Knoten. Ein weiterer Grund für gute Ergebnisse in dieser Disziplin ist, dass beim Bulkloading keine Änderungen an Records auftreten, die deren Größe verändern[8]. Somit ist eine einmal vom Recordmanager getroffene Platzierungsentscheidung auch nach einiger Zeit immer noch gut. Beim Baumaufbau mit Splitalgorithmen wachsen demgegenüber die meisten Records erst nach und nach, wodurch viele Verweisrecords im Recordmanager entstehen. Hierdurch geht zum einen die Anzahl an Zugriffen hoch und zum anderen die SPAN runter.

[8] Änderungen der Elternidentifikatoren beeinflussen die Größe eines Records nicht.

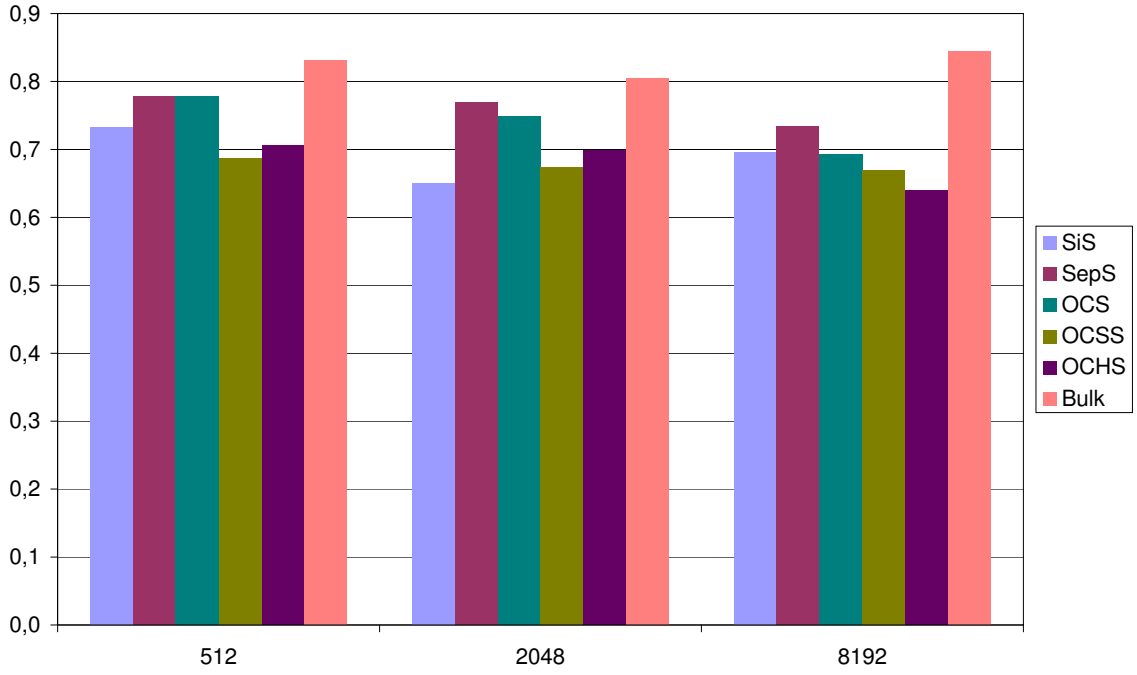


Abbildung 5.19: Speicherplatzausnutzung der verschiedenen Verfahren in Abhängigkeit zur Blockgröße. P=16384

Die Speicherplatzausnutzung ist besonders hoch bei denjenigen Verfahren, die viele kleine Records erzeugen. Die Abhängigkeit von der Blockgröße ist demgegenüber generell eher gering.

Es gibt auch Unterschiede je nach der verwendeten Strategie des Recordmanagers (siehe Kapitel 3.2.3.2). In den Abbildungen 5.20 und 5.21 sind die Zugriffszahlen für den Baumaufbau mit den zwei besten Splitalgorithmen OCSS und SepS dargestellt. Bei dieser Anfragelast ist die hybride HY(10,0.85)-Strategie deutlich im Vorteil. Diese Anfragelast mit sehr vielen Änderungsoperationen und wachsenden Records scheint dieser Strategie entgegen zu kommen. Zu Bedenken ist hier jedoch, dass die SPAN bei dieser Strategie etwa 5% schlechter ist, als bei der BFENFH(10)-Strategie. Die AO-Strategie kann bei der Anzahl der Zugriffe mit der hybriden Strategie mithalten, jedoch ist die Speicherplatzausnutzung bei ihr, wie zu erwarten war, noch einmal erheblich schlechter.

Mit den Diagrammen lässt sich außerdem eine bereits getätigte Aussage verifizieren, und zwar die Aussage über die optimale Blockgröße B. Für beide Verfahren beträgt diese 512 Bytes. Erstaunlich ist, dass die Verfahren bei 2

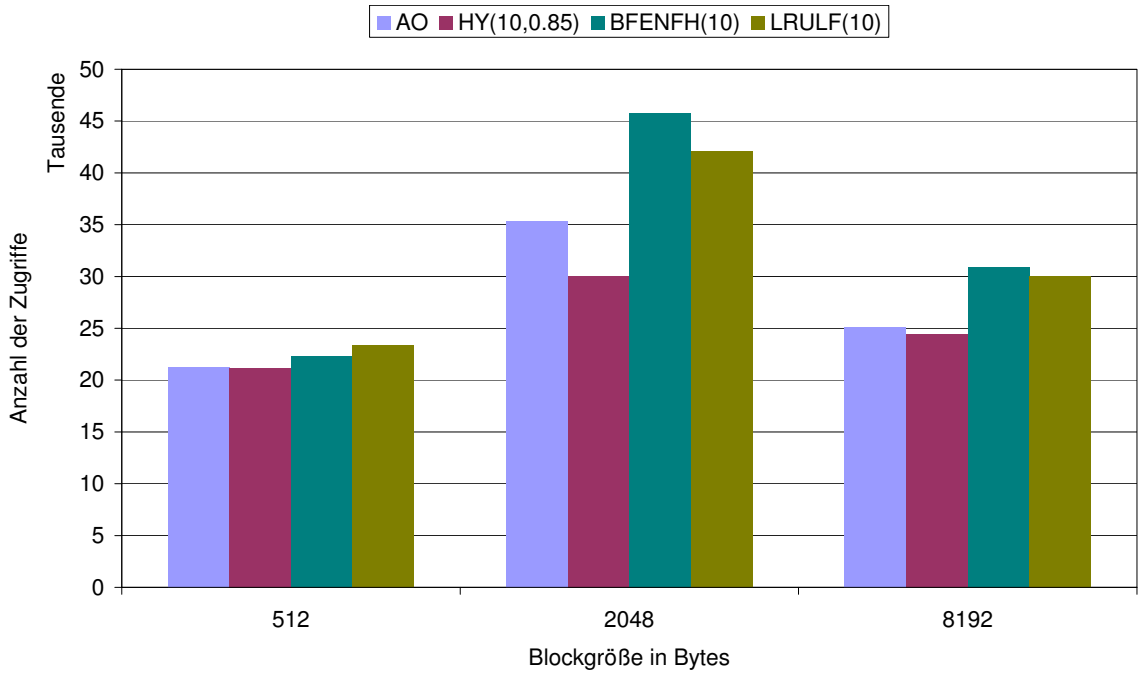


Abbildung 5.20: Anzahl der Zugriffe für den Baumaufbau mit der OCSS-Strategie bei verschiedenen Strategien des Recordmanagers und verschiedenen Blockgrößen. P=16384

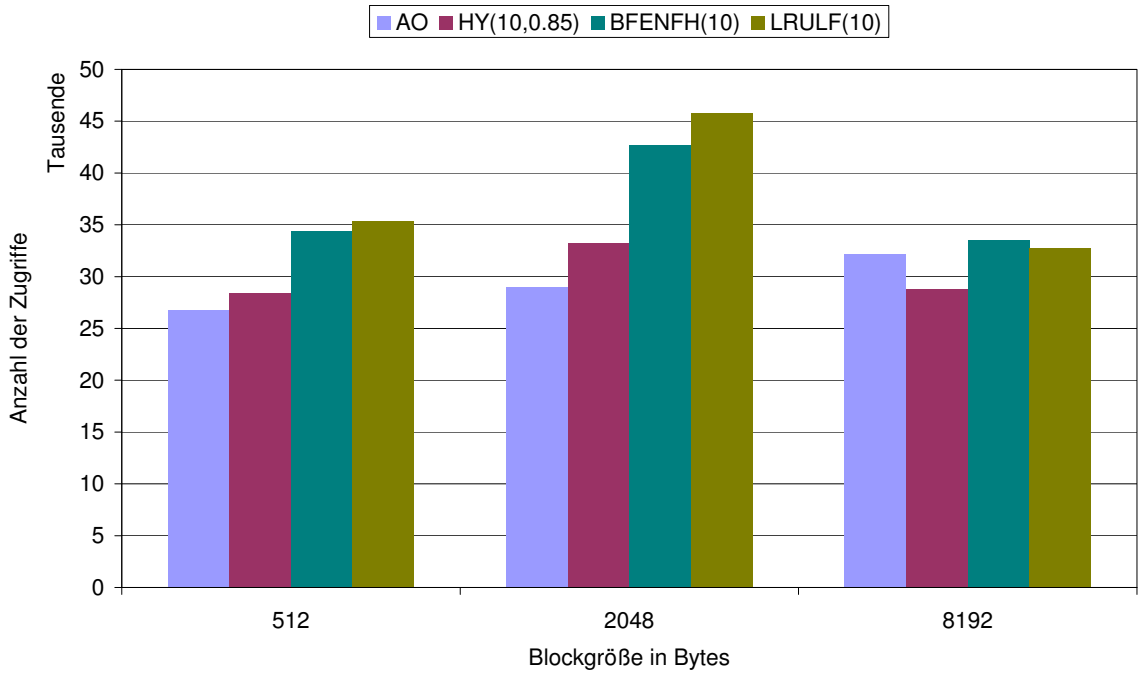


Abbildung 5.21: Anzahl der Zugriffe für den Baumaufbau mit der SepS-Strategie bei verschiedenen Strategien des Recordmanagers und verschiedenen Blockgrößen. P=16384

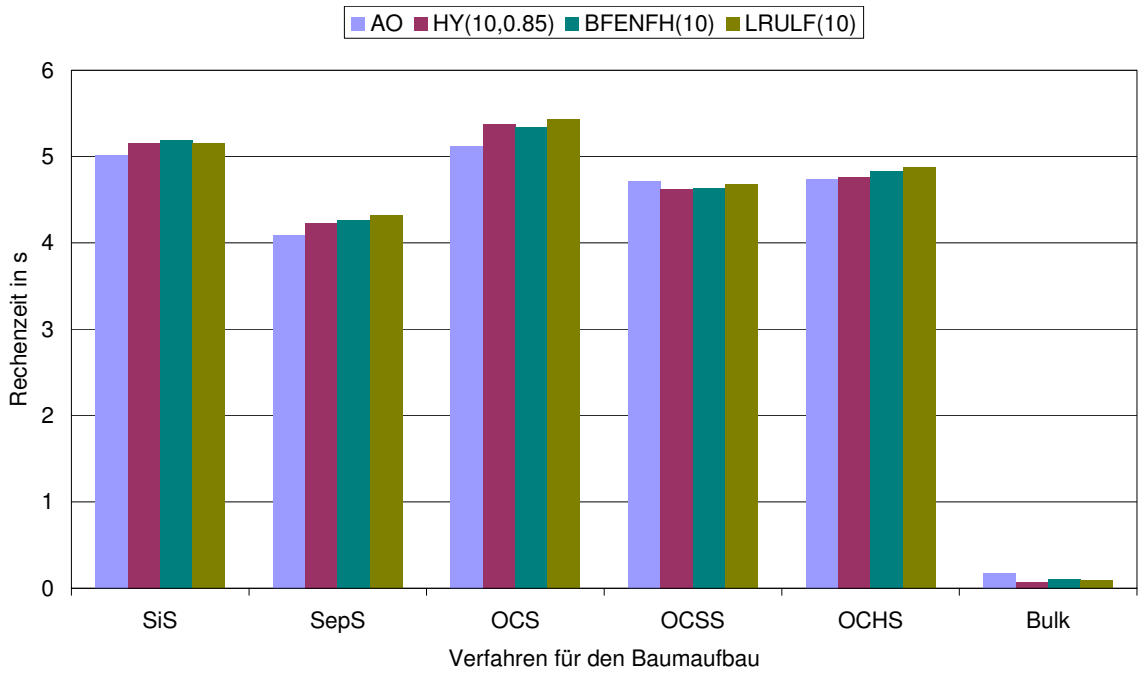


Abbildung 5.22: Rechenzeit für den Baumaufbau in Abhängigkeit vom Verfahren und von der Strategie des Recordmanagers. B=512, P=16384

KB großen Blöcken einbrechen, allerdings bei 8 KB Blöcken wieder deutlich schneller werden. Dies lässt sich auf Merkmale des XMark1-Dokuments zurückführen. Viele Teilbäume des Dokuments besitzen eine Größe knapp über der 2KB Grenze, wodurch eine niedrigere oder eine höhere Blockgröße bessere Ergebnisse liefern.

Abbildung 5.22 zeigt den CPU-Aufwand für den Baumaufbau für verschiedene Verfahren und verschiedene Recordmanagerstrategien. Es ist zu sehen, dass die Strategie des Recordmanagers an dieser Stelle keine bestimmende Größe ist. Dagegen ist eine Abhängigkeit vom Splitalgorithmus deutlich erkennbar. Erwähnenswert ist, dass die Natix Splitstrategie in dieser Kategorie am besten abschneidet. Lediglich Bulkloading besitzt einen geringeren CPU-Aufwand.

Im Vergleich zu der Anzahl an Externspeicherzugriffen ist die CPU-Belastung jedoch minimal. Für die etwas über 20.000 Externspeicherzugriffe des besten Verfahrens sind bei der verwendeten Festplatte mehr als 150 Sekunden einzukalkulieren. Die CPU-Zeit von vier bis sechs Sekunden spielt im Vergleich dazu keine Rolle. Sie ist nur dann von Bedeutung, wenn auf einem Server gleich-

zeitig sehr rechenintensive Operationen ablaufen müssen, die möglichst nicht gebremst werden sollen.

5.4.8.5 Tests mit weiteren XML Dokumenten

Die Tests mit dem größeren Dokument $XMark2$ und mit dem Dokument von Shakespeare brachten ähnliche Ergebnisse. Die Ergebnisse für den Baumaufbau sind in den Abbildungen 5.23 bzw. 5.24 dargestellt.

Bei beiden Messreihen ist hauptsächlich eine Beobachtung zu machen. Der Element Split kommt hier den anderen Splitverfahren sehr viel näher als zuvor. Beim Dokument *Sha* liegt dies an der flacheren Struktur des Dokuments. Derselbe Effekt tritt bei $XMark2$ auf, da dieses im Vergleich zu $XMark1$ im Wesentlichen breiter ist.

Bei beiden Dokumenten ist außerdem Bulkloading noch vorteilhafter als zuvor bei $XMark1$. Die Zugriffszahlen bei Bulkloading sind so gering, dass die Balken in den Grafiken kaum zu erkennen sind. Dies kann auf die Probleme der Splitalgorithmen mit sehr breiten Dokumenten zurückgeführt werden. Bulkloading bleibt trotz der Breite der Dokumente nahezu genauso effizient.

5.4.8.6 Tests ohne einen Recordmanager

Eine weitere Messreihe soll ermitteln, in wieweit die verwendete Speicherungshierarchie optimal ist. Anstatt eines Recordmanagers kann immer auch ein `BlockFileContainer` oder ein `RawAccessContainer` eingesetzt werden. Hierdurch ändern sich einige wesentliche Parameter.

Die Identifikatoren der blockbasierten Container sind 2 Bytes kürzer. Somit sind Proxyknoten ebenfalls 2 Bytes kürzer. Weiterhin kann in den blockbasierten Containern ein Block der Größe B abgelegt werden, während in einem Recordmanager die maximale Recordgröße $maxRecordSize$ kleiner als B ist (wegen der benötigten Verwaltungsinformationen für die Records, die sich in einem Block befinden). In XXL gilt $maxRecordSize = B - 9$.

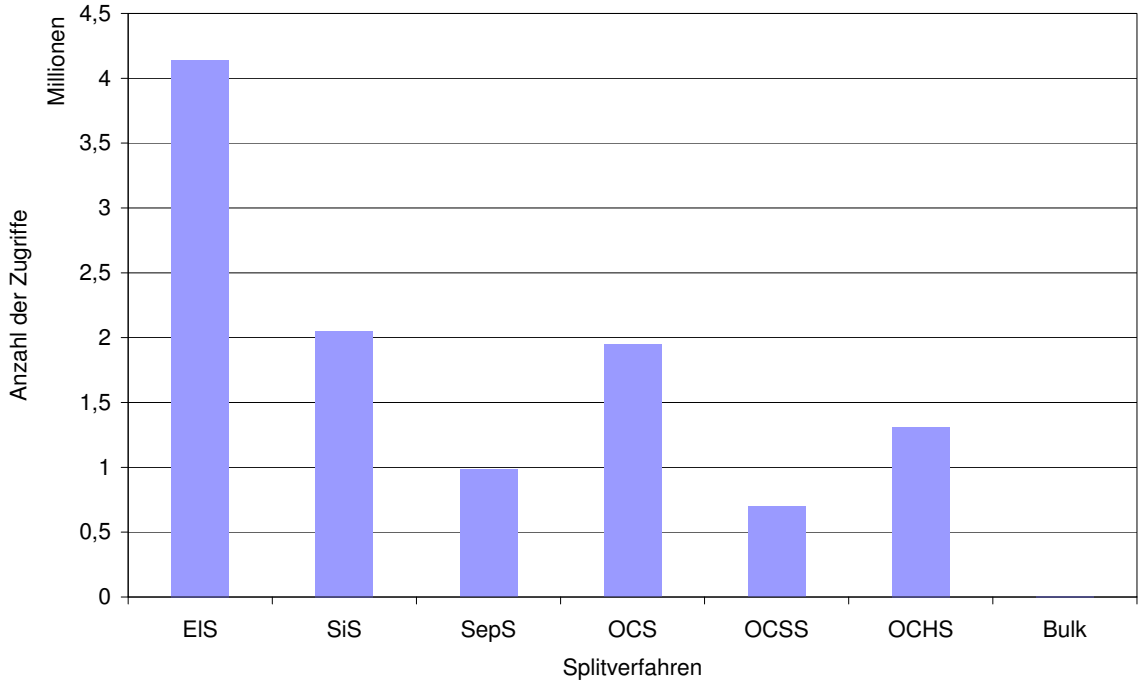


Abbildung 5.23: Anzahl der Zugriffe für den Baumaufbau mit den verschiedenen Splitalgorithmen. d=XMark2, B=512, P=163840

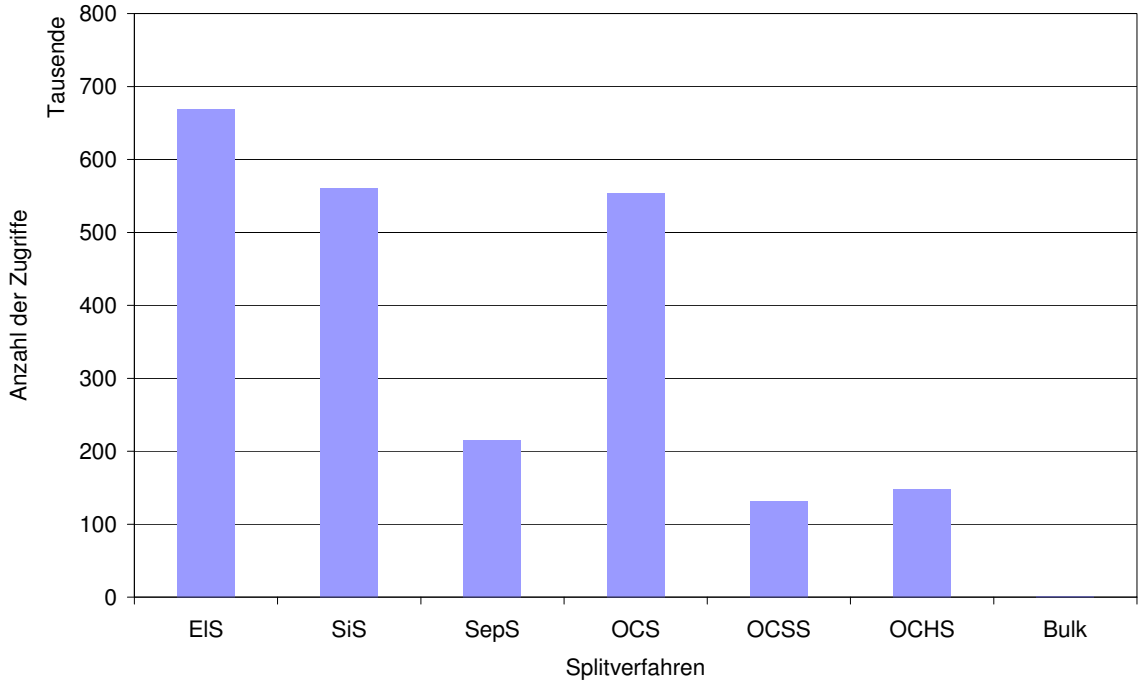


Abbildung 5.24: Anzahl der Zugriffe für den Baumaufbau mit den verschiedenen Splitalgorithmen. d=SHA, B=512, P=16384

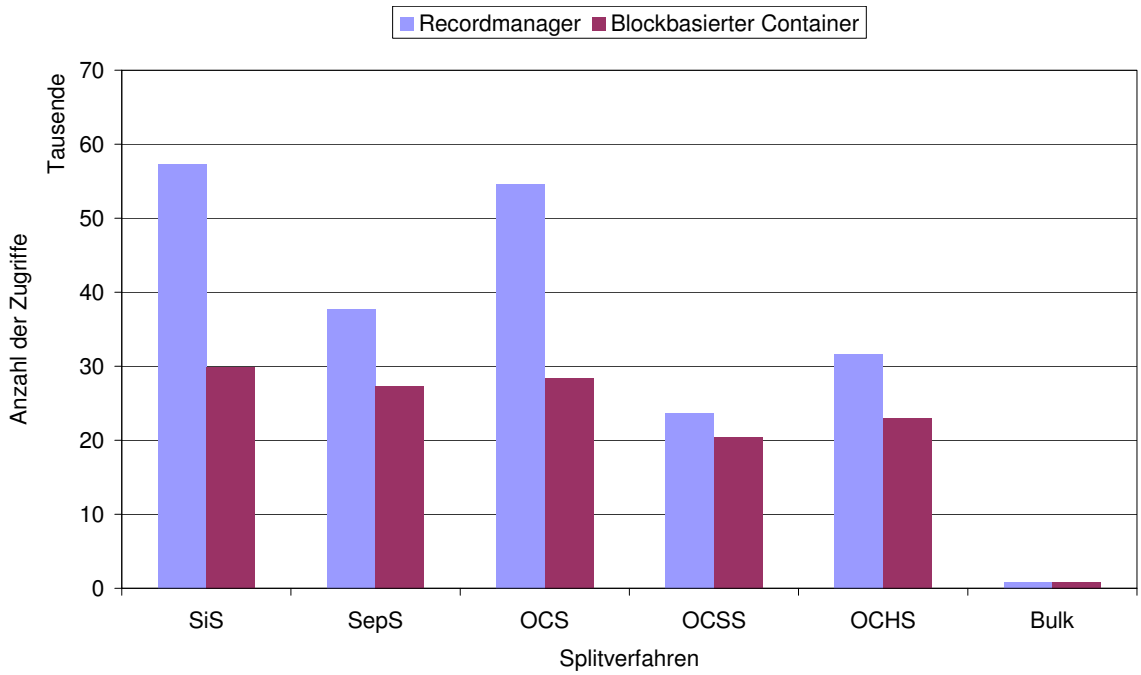


Abbildung 5.25: Anzahl der Zugriffe für den Baumaufbau mit den verschiedenen Verfahren unter Benutzung eines Recordmanagers bzw. eines blockbasierten Containers. d=XMark1, B=512, P=16384

Verwendet man ausschließlich einen blockbasierten Container, so belegt jedes Record, das die Applikation speichert, immer die volle Blockgröße B. Hierdurch liegt die Problemgröße häufig weit über der Problemgröße bei Verwendung eines Recordmanagers.

In Abbildung 5.25 sind die Zugriffszahlen für den Baumaufbau bei den effizienten Splitverfahren und Bulkloading aufgetragen[9]. Bei diesen Verfahren liegt die Anzahl der Zugriffe bei Verwendung eines blockbasierten Containers deutlich unter der Anzahl der Zugriffe bei Verwendung eines Recordmanagers. Dies liegt daran, dass beim betrachteten Szenario wenig Lokalität vorhanden ist. Bei erhöhter Lokalität würde der Recordmanager im Verhältnis besser abschneiden.

Bei der Anfrageleistung (hier nicht in einem Diagramm dargestellt) ergibt sich im Vergleich zur Einfügeleistung nur ein nennenswerter Unterschied, und zwar

[9] ElS schneidet hier sehr schlecht ab, da es extrem kleine Records in den Blöcken abspeichert und hierdurch sehr viel Speicherplatz unbenutzt lässt.

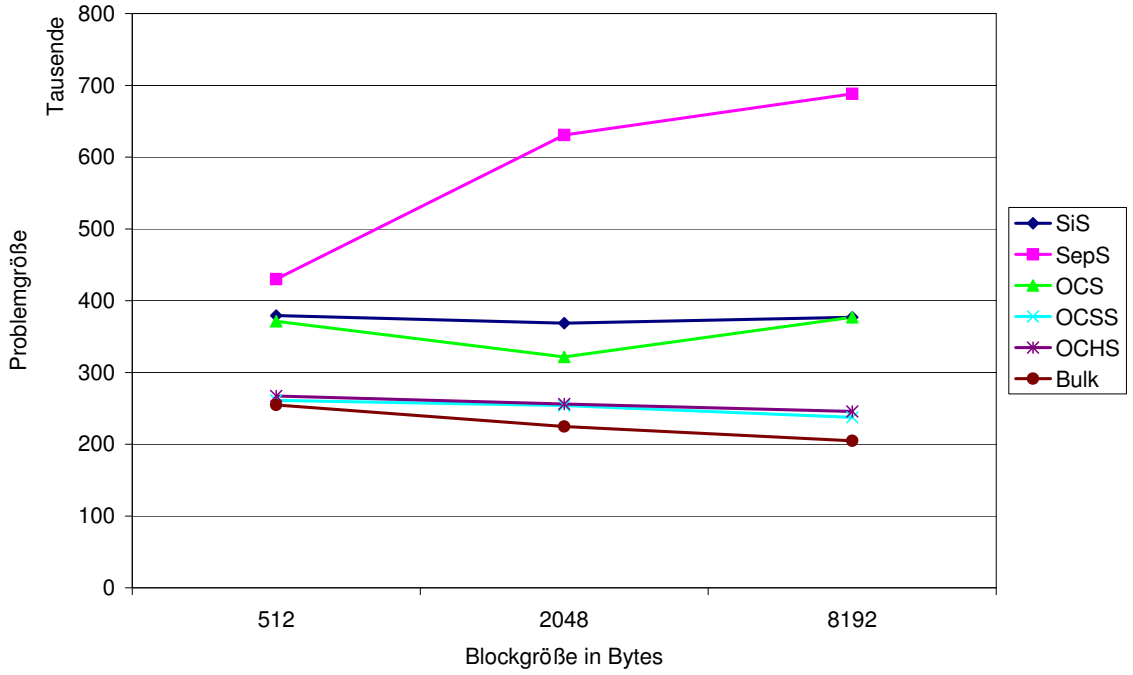


Abbildung 5.26: Problemgröße in Bytes für die verschiedenen Verfahren unter Benutzung eines blockbasierten Containers. d=XMark1

fällt SepS hinter die anderen effizienten Verfahren zurück. Der Grund hierfür ist, dass das Verfahren häufig sehr kleine Teilbäume erzeugt.

Aus demselben Grund steigt auch die Problemgröße bei SepS unter Verwendung eines Blockcontainers stark an (siehe Abbildung 5.26). Es lässt sich sagen, dass alle Verfahren, die häufig kleine Teilbäume erzeugen, eine hohe Problemgröße besitzen. Nur die Splitalgorithmen OCSS, OCHS und das Bulkloading erreichen akzeptable Werte in dieser Disziplin. Den direkten Vergleich der Problemgröße zwischen Recordmanager und Blockcontainer zeigt Abbildung 5.27. Hier ist zu sehen, dass nur die OCSS-Strategie für einen Blockcontainer geeignet ist.

Betrachten wir als letztes noch die CPU-Last. Sie ist in Abbildung 5.28 dargestellt. Hier ist kein großer Unterschied zwischen Recordmanager und Blockcontainer festzustellen. Im Schnitt sind die Blockcontainer wenige Prozente schneller, wobei dies allerdings vom verwendeten Baumaufbauverfahren abhängt. Im Verhältnis zur Anzahl der Externspeicherzugriffe spielt die CPU-Belastung erneut keine wesentliche Rolle. Sie kann somit im Vergleich zur Gesamtlaufzeit vernachlässigt werden.

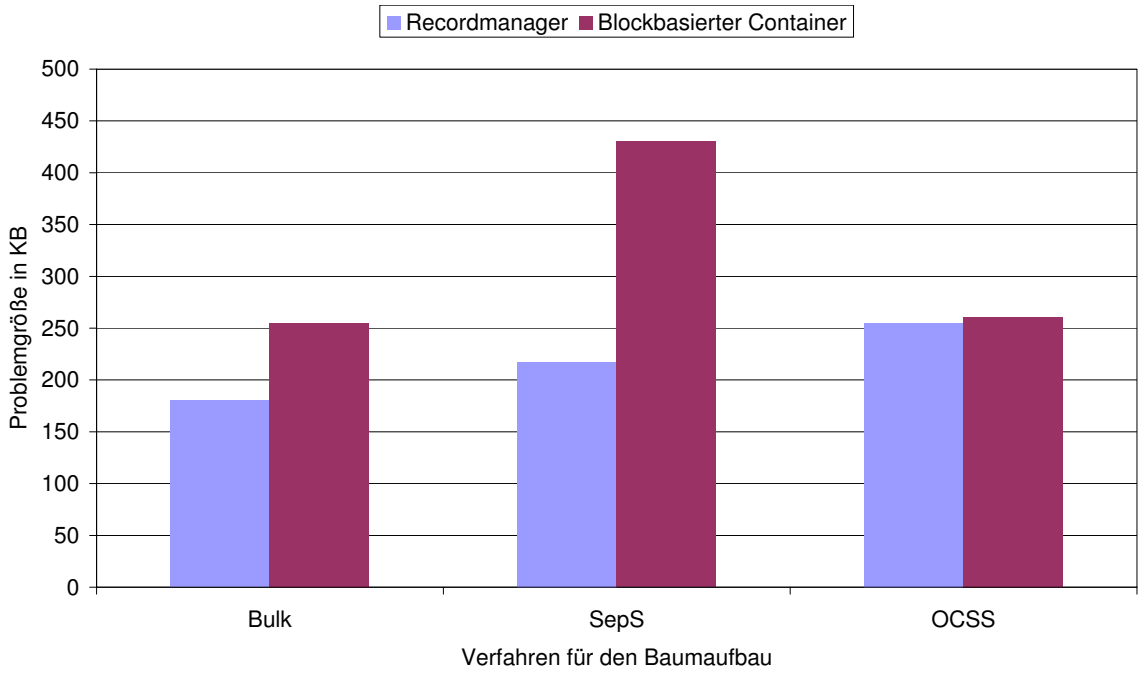


Abbildung 5.27: Vergleich der Problemgröße bei Verwendung eines Recordmanagers bzw. eines blockbasierten Containers. d=XMark1

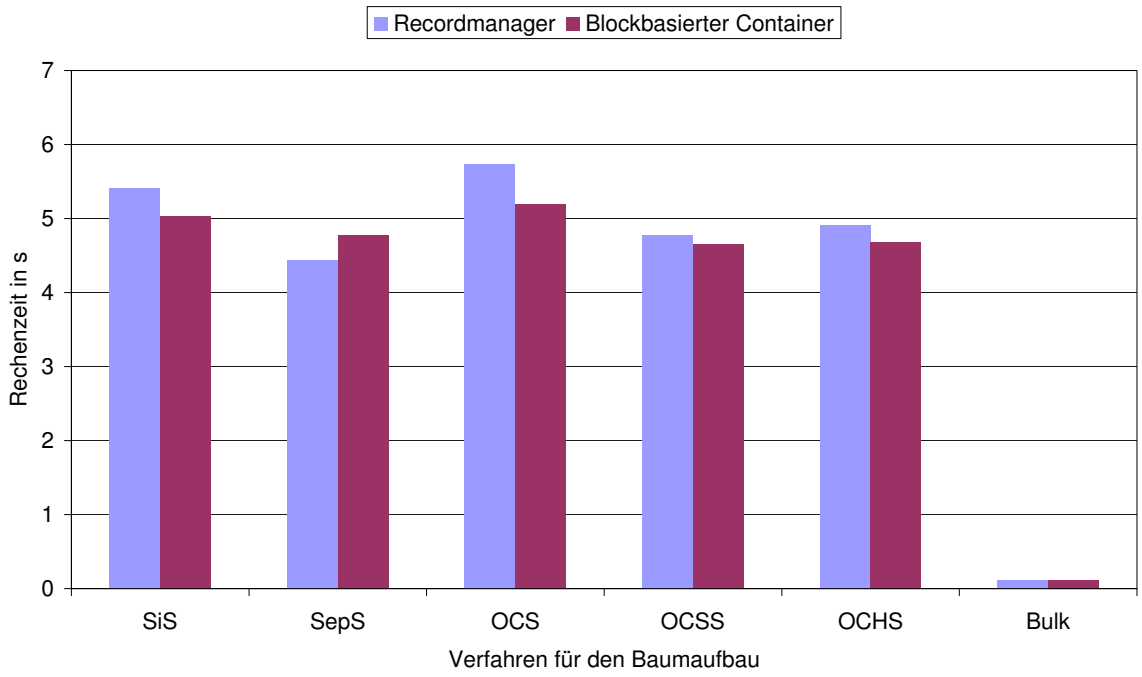


Abbildung 5.28: Rechenzeit für den Baumaufbau mit den verschiedenen Verfahren unter Benutzung eines Recordmanagers bzw. eines blockbasierten Containers. d=XMark1, B=512, P=16384

Das Fazit dieser Messreihe ist also, dass sich durch das Weglassen des Recordmanagers die Anzahl der Zugriffe verringern lässt. Allerdings tritt der Nebeneffekt auf, dass die Problemgröße hierdurch teilweise sehr deutlich ansteigt.

Das einzige Verfahren, bei dem die Problemgröße fast identisch bleibt, ist der Onecut Split mit Scaffold. Bei diesem Splitalgorithmus kann also auf einen Recordmanager verzichtet werden und stattdessen bedenkenlos ein blockbasierter Container eingesetzt werden. Die Anzahl der Zugriffe lässt sich hierdurch bei praktisch keinen nachteiligen Effekten um weitere 16.0% reduzieren.

Bei nahezu allen anderen Verfahren, insbesondere beim Natix Split (SepS), scheidet die Benutzung eines blockbasierten Containers wegen der stark erhöhten Problemgröße aus. Somit hat sich in diesen Fällen die in Natix vorgeschlagene Architektur mit Recordmanager bewährt.

5.4.8.7 Der Objektpuffer

In XXL können Puffer beliebig vor bzw. nach jeden Container geschaltet werden. Somit ist es möglich, einen Objektpuffer vor den `ConverterContainer` zu schalten, um teure Konvertierungen zwischen Java-Hauptspeicherbäumen und der Externspeicherrepräsentation zu vermeiden.

Hierbei gibt es ein größeres Problem. Die Größe des Objektpuffers muss ebenso wie die Größe des Blockpuffers in Bytes angegeben werden, da die Zuteilung von Speicher durch das XML Datenbanksystem auf dieser Basis abläuft. Aber: Die Größe eines Records entspricht nicht der Größe des Java-Hauptspeicherbaums. Beispielsweise belegen Referenzen in Java im Hauptspeicher 8 Bytes. Bei der Externspeicherrepräsentation werden hierfür aber nur noch 2 Bytes benötigt, da dies für die eindeutige Identifikation eines Knotens innerhalb eines Records ausreicht[10]. Das Problem ist, dass die Bestimmung der Hauptspeichergröße eines Objekts und erst recht die Bestimmung der Hauptspeichergröße eines Teilbaums in Java nicht direkt unterstützt wird. Eine korrekte Berechnung kann nur durch Traversierung der Strukturen mittels der *Reflection-API* erfolgen. Diese Methode ist allerdings nicht performant und kann daher in einem Produktivsystem nicht eingesetzt werden.

[10] Dies ist ausreichend für eine maximale Recordgröße von etwa 1MB.

Zur Lösung des Problems wurde untersucht, welche Beziehung zwischen der Hauptspeichergröße eines Teilbaums und der Größe der Externspeicherdarstellung besteht. Hierzu wurden bei Testläufen beide Größen für alle jemals aufgebauten Teilbäume ermittelt. Die Ergebnisse sind in Abbildung 5.29 dargestellt.

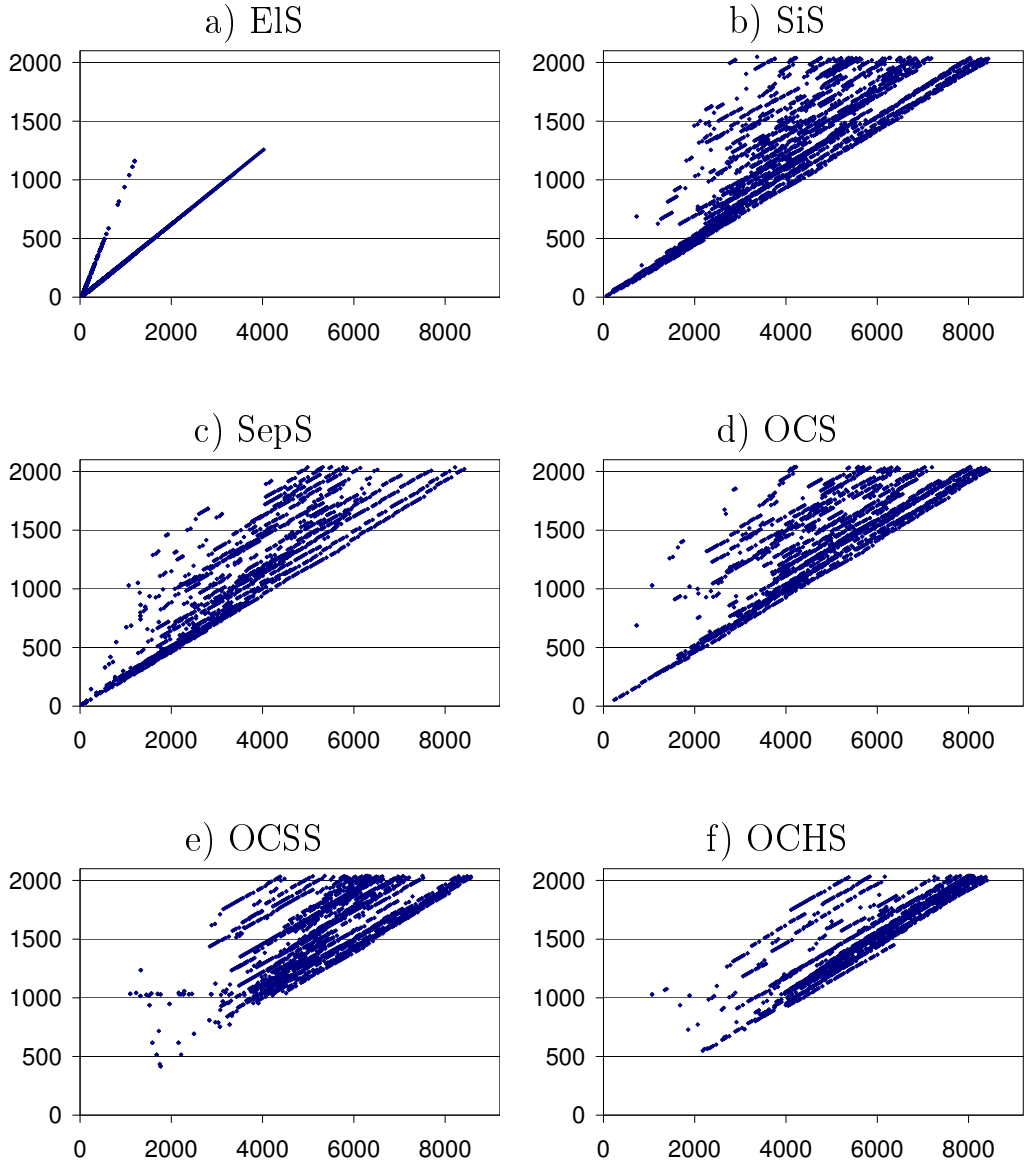


Abbildung 5.29: Größe der betrachteten Teilbäume in Java (x-Achse) und ihre serialisierte Größe auf dem Externspeicher (y-Achse). B=2048

Es zeigt sich, dass es deutliche Unterschiede zwischen den verschiedenen Splitalgorithmen gibt. Zum einen ist hier gut zu sehen, welche Art von Records die

Splitalgorithmus	XMark1	Sha
ElS	3.12	2.94
ElSA	3.08	2.94
SiS	3.56	2.97
SepS	3.58	3.17
OCS	3.62	2.97
OCSS	3.65	3.09
OCHS	3.65	3.05
Mittelwert	3.47	3.02

Tabelle 5.4: Mittlerer Faktor zwischen Java Teilbaumgröße und der Serialisierung für verschiedene Datenmengen und Splitalgorithmen

jeweiligen Verfahren erzeugen. Die OCSS und OCHS Algorithmen erzeugen nur wenige kleine Teilbäume. Die Größe der meisten Teilbäume beträgt mindestens eine halbe Recordgröße, was ja ein Ziel beim Entwurf dieser Algorithmen war.

Beim Element Split sind zwei Geraden zu erkennen. Die steil ansteigende Gerade repräsentiert ausschließlich Textknoten, für die in etwa ein Verhältnis von 1:1 zwischen Java und $\|T(k)\|$ gilt. Bei den Knoten, die auf der zweiten Gerade liegen, ist das Verhältnis etwa 1:3.3. Die Teilbäume, die von anderen Splitalgorithmen erzeugt werden und beliebige Knotenarten miteinander kombiniert enthalten, zeigen variierende Größenverhältnisse. Die Bandbreite geht von 1:1 bis zu 1:4.5.

Da in jedem Knoten die Größe der Externspeicherdarstellung bekannt ist, kann aus dieser durch Multiplikation mit einem Faktor die Java-Teilbaumgröße geschätzt werden. Die hierzu verwendeten Faktoren wurden für jeden Splitalgorithmus und verschiedene Datenmengen bei Testläufen ermittelt. Sie sind in Tabelle 5.4 aufgeführt. Es zeigt sich, dass eine deutliche Abhängigkeit von der Datenmenge existiert. XMark1 besitzt im Mittel einen um 15% höheren Faktor als Sha. Eine nennenswerte Abhängigkeit von der Blockgröße konnte demgegenüber nicht festgestellt werden.

Bei den folgenden Tests wird die Größe der Hauptspeicherstrukturen immer durch Multiplikation von $\|T(k)\|$ mit dem für das jeweilige Szenario vorbe-

rechneten Faktor abgeschätzt. Ein reales XML Datenbanksystem hat allerdings nicht die Möglichkeit, den mittleren Größenfaktor exakt zu bestimmen. Hier muss ein fester Faktor gewählt werden. Ist der Faktor zu klein, so beansprucht der Objektpuffer mehr Speicherplatz, als ihm eigentlich zugeteilt war, was einen negativen Einfluss auf die Systemleistung haben kann. Ist der Faktor zu groß gewählt, so wird der Objektpuffer immer unwirksamer, obwohl eventuell nicht der komplette Pufferplatz ausgeschöpft wurde.

Wenn das System komplett sichergehen will, dass nicht mehr Speicher verbraucht wird, als es zuteilt, so kann es mit dem maximal möglichen Faktor von 4.5 arbeiten, was jedoch negative Folgen für die Effizienz des Objektpuffers besitzt. In Zukunft könnten Verfahren entwickelt werden, die effizient bessere obere Schranken für die Hauptspeichergröße eines Teilbaums berechnen.

Als nächstes soll die Wirksamkeit der Objektpuffer anhand einer Messreihe überprüft werden. Der Speicherplatz g für Objekt- und Blockpuffer zusammen wurde auf 8 bzw. auf 16 KB gesetzt. Die Größe des Objektpuffers g_o wurde in 8 KB Schritten variiert. Entsprechend ergibt sich die Größe des Blockpuffers g_b aus $g_b = g - g_o$. Bei den Tests musste die Blockgröße fest auf 512 Bytes gesetzt werden, weil der Objektpuffer, wenn er benutzt wird, mindestens zwei Teilbäume speichern können muss[11]. Zwei Teilbäume würden bei einer Blockgröße von 2048 Bytes nach der obigen Schätzung bereits bis zu 14952 Bytes belegen, so dass keine 8 KB Schritte der Größe des Objektpuffers möglich gewesen wären.

Für eine bestimmte Speicherzuteilung an Block- und Objektpuffer wurde mit allen implementierten Verfahren die XML Speicherungsstruktur aufgebaut. Von den gemessenen, für den Aufbau nötigen Externspeicherzugriffe wurde der Mittelwert berechnet. Die Messwerte sind in Abbildung 5.30 dargestellt.

Es zeigt sich, dass der Blockpuffer die Anzahl an Zugriffen am wirksamsten reduziert. Wird allerdings teilweise oder ausschließlich ein Objektpuffer eingesetzt, so steigt die Anzahl an Zugriffen merklich an. Im schlimmsten Fall werden mehr als doppelt so viele Zugriffe benötigt. Bei einer Blockpuffergröße von 8 KB wirkt sich ein zusätzlicher Objektpuffer von weiteren 8 KB in der Anzahl der Zugriffe praktisch nicht aus.

[11] Beim Einfügealgorithmus muss genau ein Objekt im Puffer fixiert werden und ein zweiter Eintrag möglich sein, damit weitere Teilbäume überhaupt gelesen werden können.

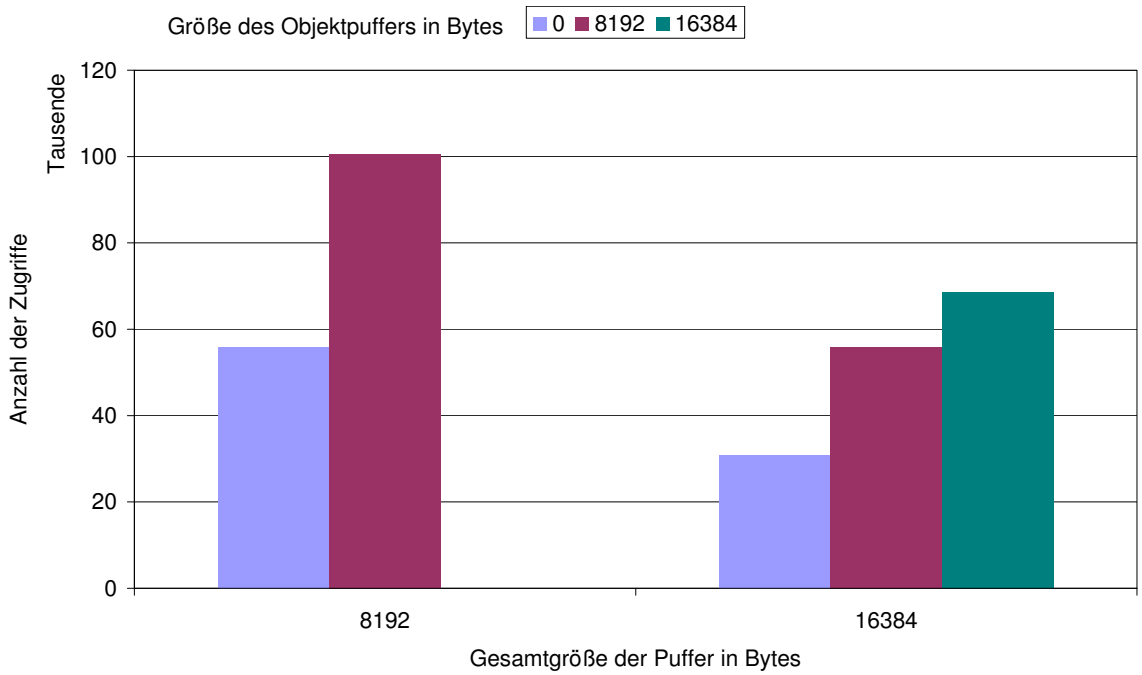


Abbildung 5.30: Anzahl der Zugriffe beim Baumaufbau mit verschiedenen Zuteilungen an Blockpuffer und Objektpuffer (Mittelwerte über die verschiedenen Verfahren). B=512

Bei der Rechenzeit ist die Situation genau umgekehrt. Je größer der Objektpuffer, desto weniger Baumkonvertierungen müssen durchgeführt werden und desto weniger Rechenzeit wird benötigt (siehe Abbildung 5.31). Allerdings kann die Verbesserung der Rechenzeit bei weitem nicht eine hohe Anzahl an Zugriffen kompensieren.

Das Fazit dieses Abschnitts ist, dass Baumkonvertierungen in der Tat sehr teuer sind und durch einen Objektpuffer vermieden werden können. Objektpuffer müssen allerdings etwa 3-4 Mal so groß sein wie Blockpuffer, um den selben Inhalt speichern zu können. Die Zuteilung des kompletten, vorhandenen Speicherplatzes an einen Blockpuffer hat im gewählten Szenario eine deutlich bessere Performanz zur Folge. Ein Objektpuffer kann allerdings von Bedeutung sein, wenn eine hohe Lokalität von Operationen im XML Baum gegeben ist.

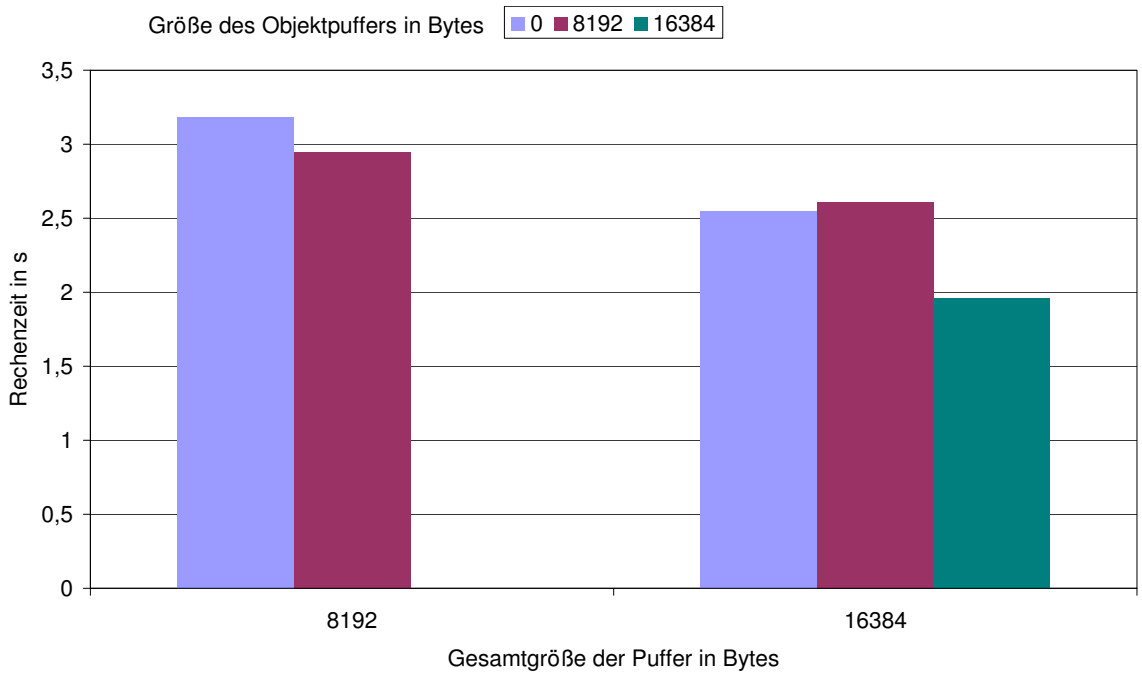


Abbildung 5.31: Rechenzeit für den Baumaufbau mit verschiedenen Zuteilungen an Blockpuffer und Objektpuffer (Mittelwerte über die verschiedenen Verfahren). B=512

5.4.8.8 Laufzeittests mit direktem Festplattenzugriff

Bislang wurden nur Tests durchgeführt, bei denen die Gesamtlaufzeit durch die Anzahl der Externspeicherzugriffe und durch den CPU-Aufwand abgeschätzt wurden. Mit dem in Kapitel 3.2.2.1 vorgestellten Mechanismus für direkten Festplattenzugriff sind jedoch auch faire Laufzeittests möglich. In diesem Abschnitt soll das bisher verwendete Modell validiert werden. Hierzu wurden einige der oben bereits durchgeführten Tests unter Benutzung des direkten Festplattenzugriffs auf dem in Kapitel 2.5 beschriebenen Testsystem nachgefahren.

Die Ergebnisse für den Baumaufbau sind in Abbildung 5.32 dargestellt. Der Vergleich mit den Externspeicherzugriffen von Abbildung 5.14 zeigt, dass es keine nennenswerten Verschiebungen gibt, also das theoretische Modell bestätigt wird.

Dasselbe Bild zeigt sich auch, wenn man die Diagramme zur Anfrageleistung miteinander vergleicht. Die Ergebnisse mit direktem Festplattenzugriff (Abbil-

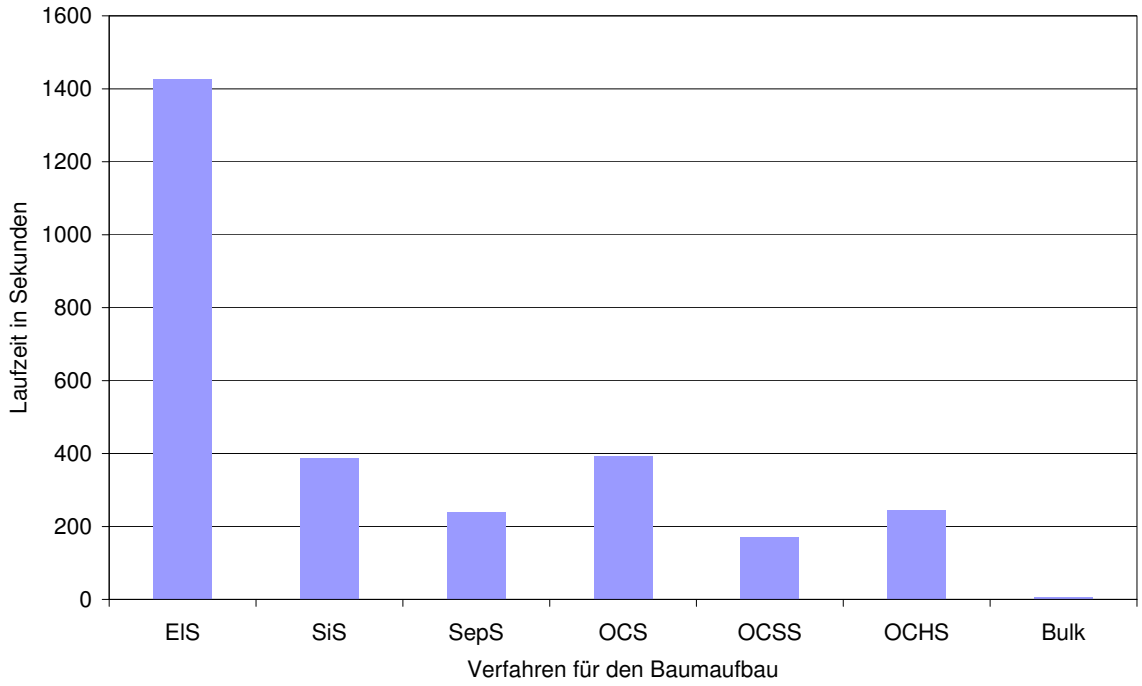


Abbildung 5.32: Laufzeit für den Baumaufbau mit direktem Festplattenzugriff für die verschiedenen Verfahren. B=512, P=16384

dung 5.33) entsprechen ziemlich genau den Ergebnissen von Abbildung 5.15, wo nur Zugriffe gezählt wurden.

Auch die optimale Blockgröße von 512 Bytes sowie andere festgelegte Parameter ergeben sich identisch aus den Laufzeitmessungen mit direktem Festplattenzugriff (nicht in Diagrammen dargestellt). Somit ist das Modell des Zählens von Externspeicherzugriffen in den gezeigten Fällen ein adäquates Maß für die tatsächliche Leistung der Verfahren.

5.4.9 Zusammenfassung und Ausblick

In diesem Abschnitt wurde eine ausführliche Evaluierung von Splitverfahren für nativen XML Speicher durchgeführt. Zum Teil handelte es sich hierbei um Tests, die eigentlich schon in Vorgängerarbeiten hätten erfolgen sollen. Beispielsweise wurden in den Arbeiten zu Natix einige Verfahren und Parameter vorgeschlagen, die dort nicht evaluiert wurden. Es konnte außerdem die Bedeutung von Bulkloading für die Leistung von Anfragen aufgezeigt werden.

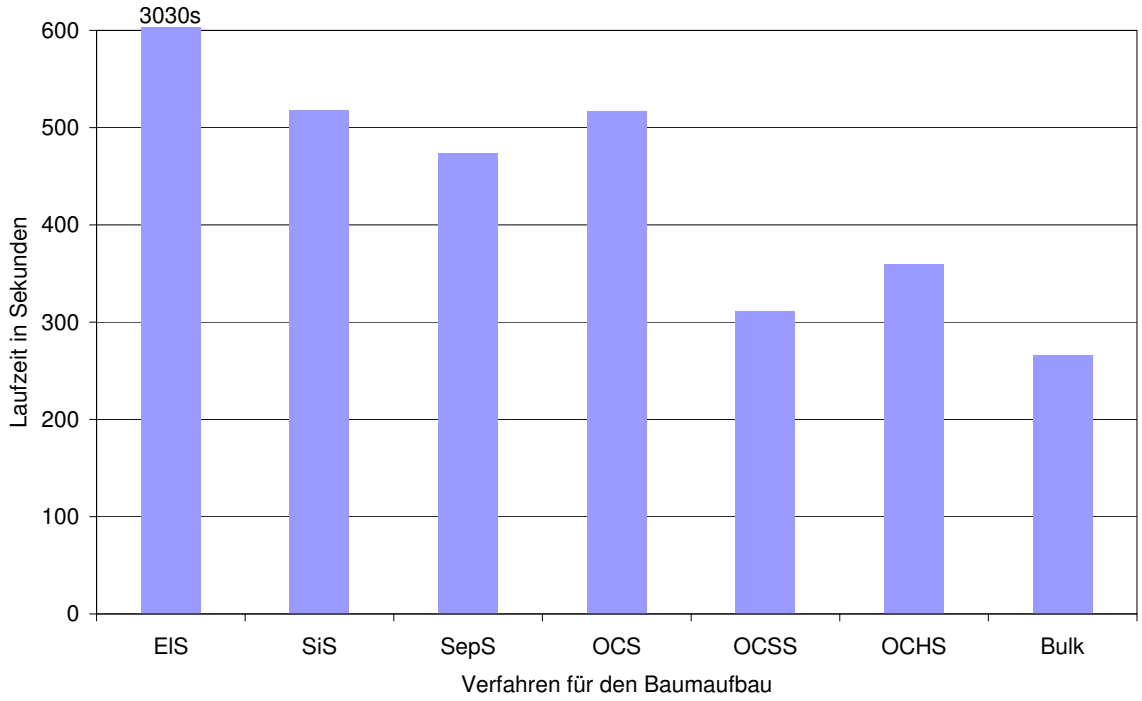


Abbildung 5.33: Laufzeit der XMark-Anfragen für verschiedene Baumstrukturen unter Verwendung des direkten Festplattenzugriffs. Die Y-Achse wurde hier passend für die schnelleren Verfahren skaliert. B=512, P=16384

Bulkloading ist somit essentiell für native XML Datenbanksysteme, und zwar sowohl für die Einfügeleistung, als auch für die Anfrageleistung.

Bei der Evaluierung der Splitalgorithmen zeigte sich der Onecut Split mit Scaffold als bestes Verfahren. Er hat sowohl eine bessere Einfügeleistung, als auch eine bessere Anfrageleistung. Durch den Verzicht auf einen Recordmanager kann die Leistung des Verfahrens weiter gesteigert werden, was bei anderen Splitalgorithmen aufgrund der vielen kleinen Teilbäume nicht möglich ist.

Ein weiteres Ergebnis ist, dass Objektpuffer nur mit Vorsicht eingesetzt werden sollten. Zur Reduzierung der Externspeicherzugriffe sind sie nicht geeignet. Sie können nur bei einer hohen Lokalität von Operationen oder bei viel zur Verfügung stehendem Hauptspeicher eine Rolle spielen.

In künftige Splitalgorithmen könnten noch weitere Ideen einfließen. Beispielsweise wäre es möglich, die Clusterung in der XML Speicherungsstruktur von

vorhandenen ID/IDREF-Beziehungen abhängig zu machen. Somit könnten XML Daten, die in Teilen Graphstrukturen besitzen, effizient unterstützt werden. Ein mögliches Ziel wäre es, kleinere Sprünge im Dokument ohne zusätzliche Externspeicherzugriffe zu ermöglichen und somit das Identifikationsattribut und die referenzierte Stelle im selben Record zu halten.

5.5 Indexstrukturen für nativen XML Speicher

Für XML wurden bereits viele verschiedene Arten von Indexstrukturen vorgeschlagen (siehe Kapitel 5.1.1.3). Die Anzahl an wirklich benötigten Indexstrukturen in einem XML System ist auch deutlich höher als in einem relationalen System. Zum einen sollen strukturbehaftete Anfragen beantwortet werden, wozu verschiedene Strukturindexe benötigt werden. Zum anderen sind Volltextindexstrukturen nötig, wie sie aus dem Information Retrieval bekannt sind. Die Fragestellung, wie Information Retrieval und strukturelle Anfragen verbunden werden können, beschäftigt derzeit viele Arbeiten [Fuh04].

Wie bereits beschrieben existieren zwei prinzipiell unterschiedliche Indexstrukturen: die referenzierenden und die nicht referenzierenden Indexstrukturen. Alle nicht referenzierenden Strukturen können sehr einfach mit nativem XML Speicher verwendet werden. Bei den referenzierenden Indexen ist dies anders. In der Literatur wurden bislang noch keine Indexstrukturen dieser Art für nativen XML Speicher vorgeschlagen. Dies hat seine Gründe, weil sich deren Anbindung als problematisch erweist, wie sich noch zeigen wird.

Im folgenden Abschnitt werden zunächst die generellen Probleme von referenzierenden Indexstrukturen erläutert. Anschließend wird der nicht referenzierende Signaturindex beschrieben, der in XXL implementiert wurde.

5.5.1 Referenzierende Indexstrukturen

Als erstes stellt sich die Frage, wie in einer nativen XML Speicherungsstruktur ein Knoten eindeutig referenziert werden kann. Wie bereits in Kapitel 5.1.1.3 erläutert, kommen XPath-Ausdrücke als logische Identifikatoren nicht in Frage.

Somit bleiben nur die Unterstützung des ID/IDREF-Konzepts oder physische Identifikatoren übrig.

Für XXL wurde eine Zwischenlösung angedacht. Eine Referenz besteht aus einem Recordidentifikator zusammen mit einer im Dokument eindeutigen Markierung. Der Recordidentifikator erlaubt das direkte Laden des entsprechenden Teilbaums und die Markierung erlaubt das Auffinden des gesuchten Knotens darin. Wir gehen im Folgenden davon aus, dass nur solche Knoten eine Indexmarkierung erhalten, die auch in Indexen referenziert werden. Dies sollte aus Effizienzgründen nur eine Minderheit der Knoten sein, da die nicht benötigten Markierungen nur unnötig Speicherplatz belegen.

Durch Änderungen im Dokument kann es nun passieren, dass solche Referenzen ungültig werden. Hier stellen sich ähnliche Probleme wie bei den in Kapitel 5.2 beschriebenen index-organized Tables. Wenn ein Split im Index durchgeführt wurde, sind die Referenzen auf Stellen im Index eventuell nicht mehr aktuell.

Der Indexmanager kann durch die eindeutige Markierung feststellen, ob eine Referenz noch aktuell ist. Ist sie dies nicht, so gibt es verschiedene Möglichkeiten, wie das Problem gelöst werden kann:

1. Der Indexmanager löscht den entsprechenden Eintrag aus dem Index und schaltet für die vorhandene Anfrage auf eine erschöpfende Suche um. Hierbei wird der Indexeintrag des Knotens (oder die Indexeinträge mehrerer betroffener Knoten) aktualisiert.

2. Der Indexmanager macht eine lokale Suche in den benachbarten Teilbäumen. Welche Teilbäume besucht werden müssen, ist dabei abhängig vom verwendeten Splitalgorithmus. Allgemein kann ein Knoten durch einen Split im XML Baum sowohl nach oben, als auch nach unten wandern. Beim Separator Split sind beide Richtungen möglich, wohingegen bei den anderen Splitalgorithmen (Element Split, Simple Split, Onecut Split) nur nach unten abgesplittet wird.

   Ist die Suche nach einer vorgegebenen Suchdistanz erfolglos, so wird sie abgebrochen und nach Punkt 1 weiter verfahren.

Falls viele Löschoperationen von Knoten erfolgt sind und diese aus Effizienzgründen nicht gleich an den Index weitergegeben wurden, so kann es sein, dass das beschriebene Verfahren leistungsmäßig schlecht abschneidet. Die Identifikatoren von gelöschten Records können zwischenzeitlich an neu erstellte Records vergeben worden sein, wodurch der Recordidentifikator im Index eventuell auf eine völlig andere Position im Dokument verweist.

3. Die dritte Möglichkeit orientiert sich an der Lösung des Problems im Cluster Index. Hier wird anstelle des direkten Zugriffs in den referenzierenden Indexen eine Suche nach einem eindeutigen Schlüssel im Cluster Index durchgeführt. Dies lässt sich auf den hiesigen Fall übertragen.

   Hierzu wird ein zusätzlicher B^+-Baum verwendet. Die Einträge in den Baum enthalten als Suchschlüssel die oben eingeführte, eindeutige Knotenmarkierung und darüber hinaus noch den Recordidentifikator, mit dem der Knoten aktuell zu finden ist. In den Splitalgorithmen werden immer dann, wenn referenzierte Knoten[12] das Record wechseln, entsprechende Einfüge- bzw. Änderungsoperationen im B^+-Baum gemacht.

   Wenn der Indexmanager einen Eintrag nicht im referenzierten Teilbaum findet, so befragt er den B^+-Baum nach dem tatsächlichen Record und bringt seinen Index anschließend wieder auf den aktuellen Stand. Wenn im Knoten selbst noch zusätzlich die Anzahl der vorhandenen Referenzierungen in Indexen gespeichert wird, sowie die Anzahl der nach einem Recordwechsel bereits aktualisierten Referenzen, so kann ein Eintrag aus dem B^+-Baum gelöscht werden, sobald alle Indexe ihren Eintrag auf den Knoten modifiziert haben (und in der Zwischenzeit kein weiterer Split erfolgt ist).

Es stellt sich nun die Frage, wie ein referenzierender Index für XPath aufgebaut sein kann. Die Suche von kurzen XPath-Ausdrücken, die bei der Wurzel beginnen, wird schon effizient von der nativen Speicherungsstruktur unterstützt[13].

[12] Referenzierte Knoten sind Knoten, die mit einer Markierung versehen sind

[13] Die native Speicherungsstruktur ist für kurze XPath-Ausdrücke eine gute Indexstruktur.

Ein XPath-Index kann also nur bei der Auswertung von längeren oder von nicht an der Wurzel startenden XPath-Ausdrücken einen Vorteil bringen.

Eine vorstellbare Indexstruktur sieht wie folgt aus: Es werden im Index Referenzen auf alle Knoten mit einem bestimmten Namen n gespeichert. Zusätzlich zu der Referenz wird immer noch der Pfad von der Wurzel bis zu den Knoten, sowie eine Positionsnummer abgelegt. Die Positionsnummer beschreibt die Position des Knotens im Dokument relativ zu den anderen referenzierten Knoten (in Dokumentenreihenfolge).

Bei dieser Struktur stellt sich die Frage, welche Knoten in den Index mit aufgenommen werden sollten. Hierfür einige Kriterien:

1. Damit der Index einen möglichst großen Vorteil gegenüber der Speicherungsstruktur erreicht, sollten die Knoten nicht zu hoch in den Dokumenten stehen.

2. Die Knoten sollten in den anfragenden Applikationen möglichst häufig in XPath-Ausdrücken verwendet werden (damit der Index möglichst häufig benutzt werden kann).

3. Es sollten möglichst wenige dieser Knoten in Dokumenten existieren, damit der Index klein bleibt. Am Besten wäre es, dass er im Hauptspeicher gehalten werden kann. Somit wären viele XPath-Anfragen mit wenigen Externspeicherzugriffen zu beantworten.

Für den Optimierer des XML Datenbanksystems stellen sich hierdurch einige neue Fragen. Wann greift er auf den Index zurück und wann benutzt er die Navigationsoperationen der XML Speicherstruktur? In welchen Fällen wird der XPath-Index um einen weiteren Knotentyp erweitert und wann wird ein Knotentyp wieder entfernt? Die Beantwortung dieser Fragen geht über das Thema dieser Arbeit weit hinaus. Hier müssen künftige Arbeiten Klarheit bringen.

Die Implementierung von referenzierenden Indexstrukturen ist in XXL bislang noch nicht erfolgt. Somit liegen auch keine Erfahrungen vor, welche der Strategien bei nicht mehr aktuellen Referenzen in der Praxis am besten funktionieren.

Dahingegen wurde in XXL bereits ein nicht referenzierender Signaturindex integriert. Dieser wird in dem nun folgenden Abschnitt vorgestellt.

5.5.2 Signaturindex

Im nativen XML Speicher von XXL wurde ein Index implementiert, der ohne Referenzierugen von Dokumentstellen auskommt. Er basiert auf Signaturen und dient dazu, dass bei der Auswertung von XPath-Anfragen möglichst wenige Blöcke besucht werden müssen. Es folgt nun eine kurze Einführung in Signaturen und deren Prinzipien. Anschließend wird die Implementierung des Indexes in XXL beschrieben und eine Evaluierung durchgeführt.

5.5.2.1 Signaturen

Signaturen sind Bitfelder einer vordefinierten festen Länge l, in denen k Bits auf Eins gesetzt sind und alle anderen Bits Null sind. k wird hierbei als Gewicht der Signatur bezeichnet. In einer Anwendung erfolgt eine Zuordnung von Objekten zu zufällig erzeugten Signaturen mit vorgegebenem l und k. Der Vorteil des Signaturkonzeptes kommt zum Tragen, sobald Mengen von Objekten betrachtet werden. Auf Signaturebene wird dies durch eine *Überlagerungsoperation* unterstützt. Die Überlagerung zweier Signaturen ist definiert als „oder"-Verknüpfung der korrespondierenden Bits (`sigOder`), was im Computer sehr effizient mit Ganzzahloperationen berechnet werden kann. Die zweite wichtige Operation auf Signaturen ist das `istIn`-Prädikat. Es bestimmt, ob eine Signatur $s1$ in einer Signatur $s2$ enthalten ist. Bei dieser Operation wird geprüft, ob jedes Bit, das in $s1$ gesetzt ist, auch in $s2$ gesetzt ist. Wenn ja, dann gibt das `istIn`-Prädikat `true` zurück, ansonsten `false`.

Sei nun P eine Menge von Objekten. Dann wird $sig(P)$ definiert als die Überlagerung der Signaturen, die den einzelnen Objekten aus P zugeordnet sind. Sei Q eine weitere Menge von Objekten, dann kann $P \subseteq Q$ nur dann gelten, wenn $sig(P)$ `istIn` $sig(Q)$ gilt (notwendige Bedingung). Somit muss der teure Vergleich der beiden Objektmengen nur dann erfolgen, wenn die schnell zu berechnende Signaturbedingung wahr ist. Signaturen besitzen also eine Filterfunktion für den Teilmengenvergleich.

Der Teilmengenvergleich wird in Datenbanksystemen beispielsweise beim Teilmengenverbund (englisch: subset join) eingesetzt. Die Berechnung des Teilmengenverbunds wäre ohne die Verwendung einer Hilfsstruktur extrem aufwendig. Eine weitere Bedeutung besitzen Signaturen in Datenbanksystemen bei Signaturbäumen [Dep86]. Diese dienen dem Treffen einer Vorauswahl, wenn bestimmt werden soll, in welchen Mengen P_i eine Menge Q vorhanden ist.

5.5.2.2 Aggregate in der XML Speicherungsstruktur

Im Rahmen dieser Arbeit wurden Signaturen als Aggregate in der XML Speicherstruktur implementiert. Zunächst soll allgemein auf Aggregate in dieser Struktur eingegangen werden.

Aggregate sind Werte, die allgemein gesagt andere Werte bezüglich eines Kriteriums zusammenfassen. In der Speicherungsstruktur in XXL wird jedem Knoten k ein Aggregat zugeordnet. Dieses muss aus dem bei k beginnenden logischen Baum (über Recordgrenzen hinweg) eindeutig zu bestimmen sein.

Ein Aggregat zu einem Objekt wird durch eine Funktion c_{agg} bestimmt, die dem Objekt einen Aggregatwert zuweist. Wenn aus zwei Aggregatwerten ein weiterer Aggregatwert berechnet werden soll, so wird die zweistellige Aggregatfunktion f_{agg} angewendet. Hierbei sollte für f_{agg} die Symmetrieeigenschaft $f_{agg}(i,j) = f_{agg}(j,i)$ gelten.

Bei Bäumen mit beliebigem Verzweigungsgrad sieht die Situation leicht verändert aus. Die Funktion f_{agg} wird in diesem Szenario ein wenig eingeschränkt. Die Parameter sind immer noch zwei Aggregatwerte, allerdings müssen diese Werte von zwei Knoten stammen, von denen der erste Knoten der Elternknoten des zweiten ist.

Zur Berechnung der Aggregate wird logisch gesehen c_{agg} zunächst auf jeden Knoten angewendet (auch innere Knoten sollen zum Aggregat beitragen können - müssen dies aber nicht). Nun wird eine Tiefensuche gestartet. Bei jedem Zurückgehen von einem Knoten zu seinem Elternknoten wird f_{agg} auf die Aggregatwerte der beiden Knoten angewendet und der berechnete Wert als Aggregat dem Elternknoten zugewiesen.

Durch diese Vorgehensweise erhält am Ende jeder Knoten einen Aggregatwert, der eindeutig ist, falls die Reihenfolge der Anwendung von f_{agg} egal war, also

$$f_{agg}(f_{agg}(i,j),k) = f_{agg}(f_{agg}(i,k),j) \text{ mit } i \text{ ist Elternknoten von } j \text{ und } k.$$

gilt[14]. Die Symmetrieeigenschaft kann für eine Baumaggregatfunktion nicht mehr gefordert werden, da der erste Parameterknoten immer der Elternknoten des zweiten Knotens sein muss und man daher die Parameter nicht vertauschen darf. Als nächstes soll ein Beispiel für solch ein Aggregat in einem XML Baum gegeben werden. Es seien:

$$c_{agg}(k) = 0$$
$$f_{agg}(a1,a2) = max(a1,a2+1)$$

Dieses Beispiel erfüllt die obigen Bedingungen für f_{agg} und führt dazu, dass in jedem Knoten die aktuelle Höhe des Teilbaums als Aggregat gespeichert wird.

Um die Berechnung von Aggregaten in der nativen XML Speicherungsstruktur zu ermöglichen, sind einige Änderungen nötig. Jedem Knoten muss im Hauptspeicher ein Aggregatwert zugeordnet werden. Auf dem Externspeicher ist nicht viel Zusatzaufwand nötig. In unserem Modell werden nur in den Proxyknoten die Aggregate persistent gespeichert. Bei deren üblicher Weise anzutreffenden Längen von 4-16 Bytes ist der hierdurch belegte Speicherplatz weitestgehend vernachlässigbar.

Änderungen in einem Teilbaum führen somit nur zu einem geringen Aufwand bei der Neuberechnung der Aggregate. Es ist nicht nötig, alle abhängigen Teilbäume komplett zu traversieren, was sehr viele Externspeicherzugriffe zur Folge hätte. Die Neuberechnung stoppt somit immer in den Proxyknoten des eigenen Teilbaums, die ja gerade die Aggregate der abhängigen Teilbäume enthalten. Somit sind bei Einfüge-, Änderungs- und Löschoperationen die notwendigen

[14] Es wäre auch möglich, die Aggregatfunktion f_{agg} auf beliebigen Knoten zu definieren und eine andere Auswertungsreihenfolge zu wählen. Dies ist jedoch zum einen nicht effizienter und zum anderen lässt die gewählte Auswertungsreihenfolge Integritätsüberprüfungen bei der Anwendung von f_{agg} zu.

Anpassungen im Wesentlichen auf das Record selbst begrenzt. Falls sich jedoch das Aggregat des Wurzelknotens des Records ändert, so müssen die Aggregate des Elternteilbaums angepasst werden, eventuell hoch bis zur Wurzel des Dokuments. Bei Einfüge- und Änderungsoperationen können diese Anpassungen bereits auf dem Weg von der Wurzel zum gesuchten Record durchgeführt werden, was etwas effizienter ist. Bei Löschoperationen ist dies nicht möglich.

Die Berechnung der Aggregate im Inneren der Teilbäume wird beim Aufbau der Teilbäume aus den Records on-the-fly durchgeführt. Wenn der Aufwand für die Auswertung der Aggregatfunktion klein ist, kann dieser Aufwand gegenüber dem Aufbau des Hauptspeicherbaums durch den Teilbaumconverter vernachlässigt werden. Bei komplexen Aggregatfunktionen wäre eventuell die persistente Speicherung der Aggregate jedes inneren Knotens oder zumindest eines Teils dieser Knoten von Vorteil, auch wenn hierdurch erhebliche Mengen an Speicherplatz in den Records belegt würden.

Die Einführung von Aggregaten bedeutet für die Splitalgorithmen einen deutlichen Mehraufwand in der Implementierung. Nach einem Split muss jeder Algorithmus sicherstellen, dass die Aggregate der einzelnen Teilbäumen korrekt sind. Ein Fehler in in den Aggregaten würde sich verheerend auf die Korrektheit auswirken, da als Folge z. B. Suchoperationen falsche Ergebnisse liefern könnten.

5.5.2.3 Signaturen als Aggregate

Für die native XML Speicherung wurde ein Aggregat bestehend aus einer Signatur verwendet:

$$c_{agg}(k) = \begin{cases} sig(k.name) & \text{falls } k \text{ ein Markupknoten ist,} \\ nullsig() & \text{sonst.} \end{cases}$$

$$f_{agg}(a1, a2) = sigOder(a1, a2)$$

Im XML Baum werden also den Knotennamen Signaturen zugeordnet und diese durch die Signaturüberlagerung aggregiert. Diese Aggregate sind von Vorteil, wenn XPath-Ausdrücke ausgewertet werden sollen. Eine XPath-Suche kann abgebrochen werden, sobald der Name eines Knotens in der XPath-Anfrage nicht unterhalb einer betrachteten Stelle im Dokument vorkommt.

Die meisten XPath-Ausdrücke bestehen in ihrer Hauptstruktur aus einer Aneinanderreihung von Knotennamen, die durch die Anwendung der `descendant` oder `descendant-or-self` Achse miteinander verbunden sind. Diese Hauptstruktur kann im Detail mit Prädikaten angereichert sein. Somit sind die folgenden Operationen auf einem XPath-Ausdruck sinnvoll:

$länge(xpath)$: berechnet die Länge der Hauptstruktur des XPath-Ausdrucks
$name(xpath, i)$: ermittelt den i-ten Knotennamen der Hauptstruktur

Darauf aufbauend lassen sich Teilsignaturen von XPath-Ausdrücken definieren. Es sei $l = länge(xpath)$. Dann ist $sig(xpath, i)$ die Signatur eines XPath-Ausdrucks von der Ebene i bis zum Ende des XPath-Ausdrucks mit

$$sig(xpath, i) = \\ sigOder(name(xpath, l), name(xpath, l-1), ..., name(xpath, i))$$

Wenn in der Anfrageverarbeitung der erste Teil eines XPath-Ausdrucks bis zum Knoten $i-1$ mit einem Dokument in Übereinstimmung gebracht wurde, so muss ein zu betrachtender Teilbaum die Signatur $sig(xpath, i)$ in seiner Signatur enthalten haben. Ansonsten kann auf das Traversieren des Teilbaums verzichtet werden.

In XXL wurde das Signaturkonzept zunächst beim Bulkloading eingebaut. Eine Implementierung in den verschiedenen Splitalgorithmen ist noch in der Entwicklung. Beim Bulkloading ist der Aufwand für die Verwaltung der Signaturen relativ gering, da der Aufbau des Dokumentenbaums sowieso von unten nach oben erfolgt (bottom-up).

Der Aufwand für Signaturanpassungen bei Einzeloperationen ist demgegenüber sehr viel größer. Hier müssen bei jeder Operation die Signaturen aktualisiert werden, und zwar beginnend mit dem betroffenen Knoten und endend an einem Knoten, dessen Signatur nach der Aktualisierung unverändert bleibt (im schlimmsten Fall die Wurzel des XML Dokuments). Hier hängt das Leistungsverhalten ganz wesentlich von der tatsächlichen Anfragelast ab, insbesondere vom Verhältnis zwischen baumverändernden Operationen zu Suchoperationen.

In Natix oder ähnlichen teilbaumbasierten Speicherungsstrukturen für XML wurde das Signaturkonzept nach meinem Wissensstand bislang nicht eingesetzt. Somit ist das beschriebene Verfahren bislang einzigartig.

5.5.2.4 Evaluierung

Bei der Evaluation wurde die größere der beiden XMark-Dateien, $XMark2$, verwendet. Die wesentlichen Parameter, die hier betrachtet werden, sind die Blockgröße B, die Länge der Signaturen s_l, sowie deren Gewicht s_g.

Zunächst wurde geprüft, wie sich die Signaturen auf den Baumaufbau auswirken. Die Messgröße war hier erneut die Anzahl an Zugriffen (wahlfrei und mehrfach). Die erste Feststellung war, dass s_g keinerlei Einfluss auf den Baumaufbau besitzt. Dies ist logisch, weil das Signaturgewicht keinen Einfluss auf die Recordgröße besitzt und außerdem die Signaturen beim Bulkloading noch nicht eingesetzt werden können, da der Baum bottom-up aufgebaut wird.

Somit ist die Anzahl an Zugriffen beim Baumaufbau nur abhängig von der Blockgröße B und von der Signaturlänge s_l. Abbildung 5.34 enthält die Messergebnisse. Wie bereits vorher dargestellt sinkt die Anzahl der Zugriffe bei wachsendem B. Eine deutliche Abhängigkeit der Einfügeleistung von s_l konnte bei den betrachteten Werten des Parameters nicht festgestellt werden. Lediglich ein leichtes Schwanken von maximal 2% ist zu bemerken, und zwar sowohl nach oben, als auch nach unten.

Auf die Problemgröße, also die Größe des `BlockFileContainer`s, hat die Signaturlänge demgegenüber einen Einfluss (Abbildung 5.35[15]). Bei $B = 512$ be-

[15] Bemerkung: Die Y-Achse beginnt in der Abbildung nicht bei 0, damit die Unterschiede deutlicher werden.

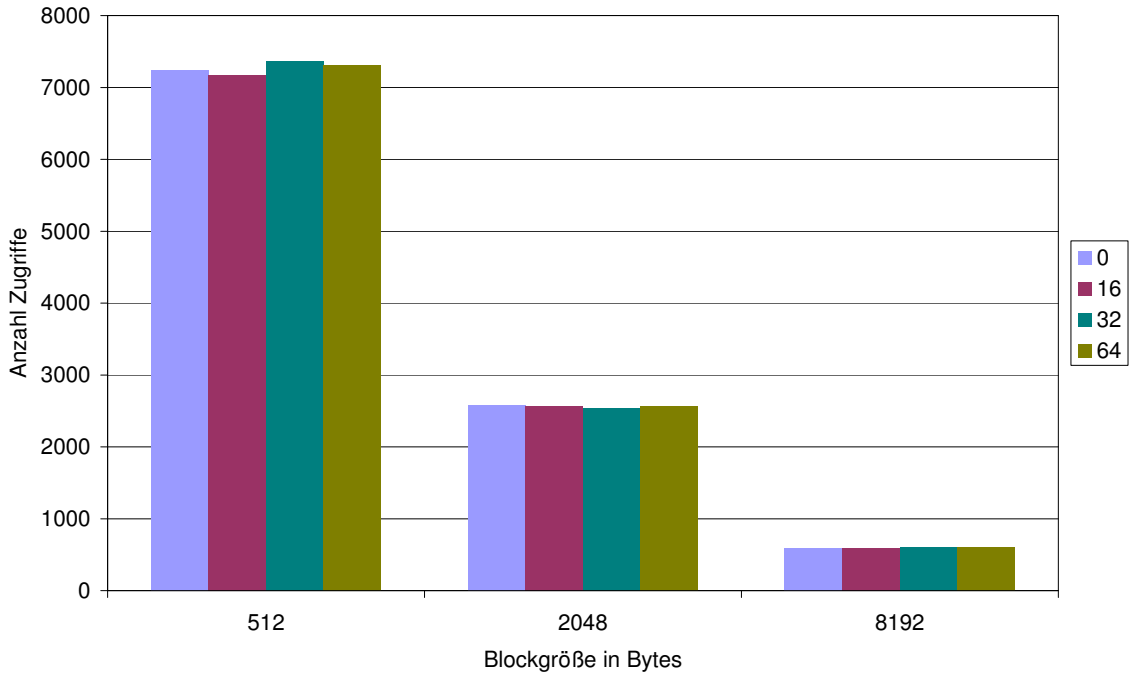


Abbildung 5.34: Anzahl der Zugriffe für den Baumaufbau mit Bulkloading unter Verwendung von Signaturen bei verschiedenen Signaturlängen (0, 16, 32 und 64 Bit). d=XMark2, P=16384.

trägt der Unterschied zwischen Baum ohne Signaturen und einer Signaturlänge von 64 Bit immerhin 2.5%. Je größer B gewählt wird, desto kleiner wird der Unterschied, weil dann deutlich weniger Proxyknoten benötigt werden.

Nun ist die Frage, wie sich der Signaturindex auf die Performanz auswirkt. Dies ist in Abbildung 5.36 dargestellt. Eine höhere Signaturlänge hat generell einen positiven Einfluss auf die Anfrageleistung. Die Wahl des Signaturgewichts ist relativ unkritisch. Ein hohes Signaturgewicht (aber kleiner gleich $s_l/2$) steigert die Anzahl der möglichen Signaturen $|S| = \binom{s_l}{s_g}$. Allerdings sorgen viele Einsen in Signaturen für eine abnehmende Indexqualität, weil dann in den oberen Indexebenen nicht ausreichend differenziert werden kann. Nach den Experimenten empfiehlt sich $s_g = 4$ als Kompromiss für alle getesteten Fälle.

Im besten Fall ($s_l = 64$, $s_g = 4$) werden bei den XMark-Anfragen 27.8% der Zugriffe im Vergleich zur Speicherungsstruktur ohne Signaturindex eingespart. Dass der Vorteil nicht noch größer ist, liegt daran, dass die Speicherungsstruktur selbst bereits eine Indexwirkung besitzt. Somit dient der Signaturindex nur der Unterstützung.

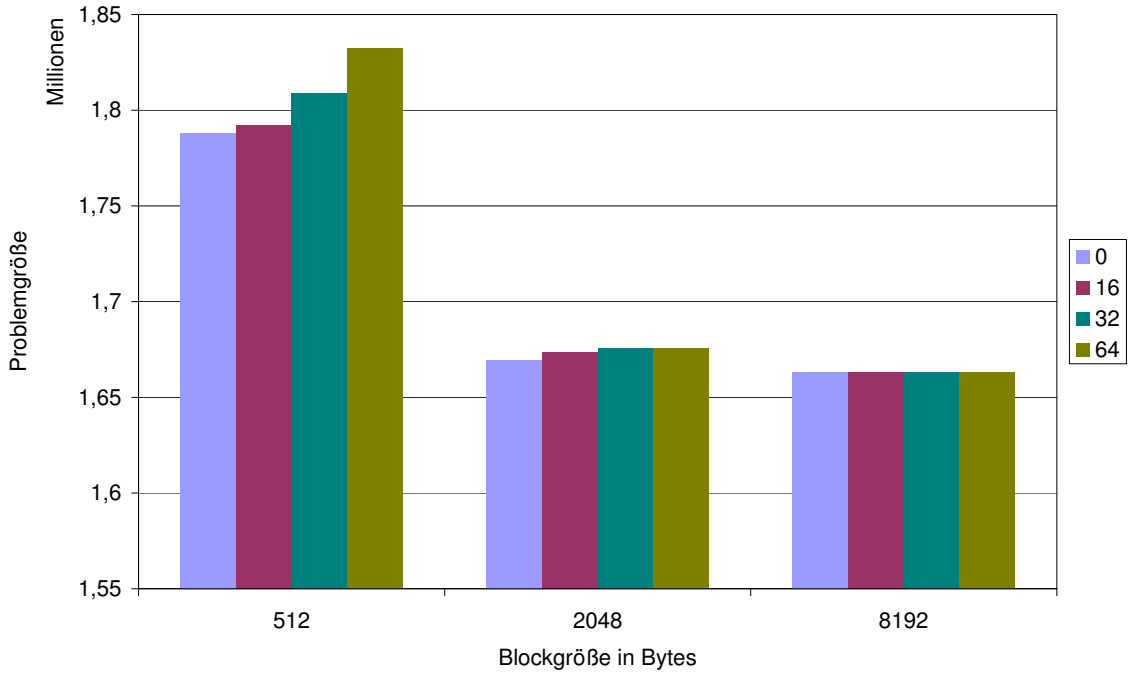


Abbildung 5.35: Größe des verwendeten `BlockFileContainers` in Abhängigkeit von B und s_l. d=XMark2.

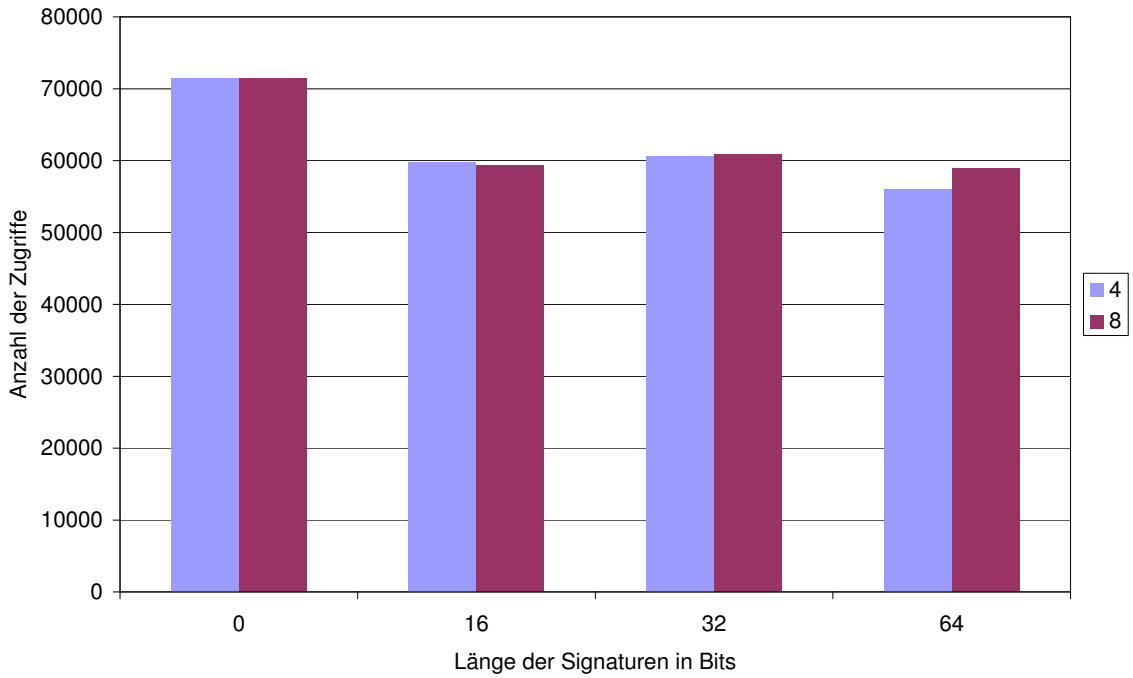


Abbildung 5.36: Anzahl der Zugriffe für die XMark-Anfragen unter Verwendung von Signaturen mit den Signaturgewichten 4 (hellblau) und 8 (lila). d=XMark2, P=16384.

Der Rechenaufwand steigt durch den Einsatz von Signaturen nur geringfügig an. Maximal wurde ein Mehraufwand von 12% gemessen. Da der reine Rechenaufwand für die 1200KB-Datei bei weniger als einer Sekunde liegt, spielt dieser im Vergleich zum Externspeicheraufwand überhaupt keine Rolle.

5.6 Verwandte Ansätze aus der Literatur

Aus der Literatur sind inzwischen einige Ansätze bekannt, die verschiedene Techniken für die Realisierung von nativem XML Speicher anbieten. Diese Ansätze werden in den folgenden Abschnitten diskutiert.

Es lässt sich feststellen, dass diese Ansätze zwar teilweise interessante Ideen enthalten, jedoch werden die Ansätze meistens nicht mit bereits bekannten nativen Speicherungsverfahren für XML verglichen. Ursache hiervon mag sein, dass die Implementierung von Referenzverfahren extrem zeitaufwendig ist. Außerdem hat sich bislang kein Standardverfahren herauskristallisiert, das überall in der Literatur als Referenzverfahren verwendet wird. Trotz der schlechten Situation, was die Evaluierung angeht, werden die entsprechenden Arbeiten auf Konferenzen akzeptiert, mit der Folge, dass neue Arbeiten wiederum keine Evaluierung gegenüber existierenden Verfahren besitzen.

5.6.1 OrientStore

OrientStore [MLLA03] bzw. OrientX [MWL+04] verfolgt einen ähnlichen Ansatz wie Natix. Zunächst wird ein Beitrag für die Klassifikation von XML Speicherungsverfahren geliefert. Es wird bei generischen Verfahren zwischen elementbasierten und teilbaumbasierten Verfahren unterschieden. Hierdurch kann die Klassifikation von [KM03] erweitert werden.

Kern des Orient-Systems sind zwei neue Splitalgorithmen, die jedoch Grammatiken voraussetzen, die bei echt semi-strukturierten Daten normaler Weise nicht vorhanden sind. Somit können diese Verfahren nicht fair mit den Splitalgorithmen dieser Arbeit verglichen werden.

Die Qualität der OrientStore Papiere lässt jedoch leider sehr zu wünschen übrig. Hierdurch kann über die Effizienz der vorgeschlagenen Splitalgorithmen wenig ausgesagt werden. Die textuelle Beschreibung der Ergebnisse widerspricht in Teilen sogar den gezeigten Grafiken und an anderer Stelle fehlen die referenzierten Grafiken sogar ganz. Bei den wenigen vorgestellten Messungen kommt heraus, dass der elementbasierte Ansatz sehr nah an die Leistung des teilbaumbasierten Ansatzes herankommt. Dies widerspricht allerdings sowohl den Ergebnissen des Natix-Papiers [KM00], als auch den Messungen, die im Rahmen dieser Arbeit gemacht wurden.

Ein weiterer Schwachpunkt der Arbeit ist, dass über die Basiskomponenten der Implementierung wenig ausgesagt wird. Insbesondere fehlen Angaben über den Recordmanager.

5.6.2 PDOM

In [HMF99] wird eine persistente DOM Implementierung vorgestellt, die auf einem speziellen, neu entwickelten Speicherungsformat basiert. Dieses legt 128 DOM-Knoten in einer Einheit innerhalb einer Datei mit wahlfreiem Zugriff ab (`java.io.RandomAccessFile`). Die Speicherungseinheiten werden mit gzip komprimiert, wodurch sie unterschiedliche Größen besitzen. Um die exakte Position einer Speicherungseinheit in der Datei zu ermitteln, wird auf eine kleine Tabelle zurückgegriffen. Über die verwendete Freispeicherverwaltung innerhalb der Datei werden leider keine Aussagen gemacht.

Für die Leistung ist von Bedeutung, dass die Knoten, die im Dokument nah beieinander liegen, auch bei der Speicherung in benachbarten Einheiten zu finden sind, also eine Clusterung vorliegt. Beim Aufbau der Struktur wird initial eine Clusterung entsprechend einem Präorder-Durchlauf durch das Dokument realisiert.

Das PDOM Speicherungsverfahren ist generell einem Element Split sehr ähnlich. Leider wird in dem Papier kein Vergleich mit dieser einfachen Strategie durchgeführt. Bei Anfragelasten, die sehr unterschiedliche Positionen im Dokument nacheinander besuchen, wird das Dekomprimieren sich negativ auf die

Gesamtleistung auswirken, so dass in diesem Fall der normale, auf einem Recordmanager basierende Element Split schneller sein wird.

Auf dem PDOM-Verfahren basiert das XML Datenbanksystem InfonyteDB [TFW03]. Seit dem ursprünglichen Artikel von 1999 hat sich das System deutlich weiterentwickelt, allerdings gibt es bislang keine neueren Angaben darüber, was an der PDOM-Ebene verändert wurde.

5.6.3 XML Query Execution Engine (XEE)

Die *XML Query Execution Engine (XEE)* [SC01] von der HU-Berlin nimmt bei der Speicherung von XML eine Trennung von Text und Dokumentstruktur vor. Der Text wird in einem *Text Array (TA)* gespeichert und die Struktur im *Access Support Tree (AST)*. Die Struktur des Dokuments wird hierbei als Metadatum zum eigentlichen Text angesehen.

Zunächst wurde die vorgeschlagene Struktur nur für die Nutzung im Hauptspeicher entworfen. In einem Folgeartikel erweiterten die Autoren ihr Konzept auf den Externspeicher [SF02]. Dieser Ansatz (AST^P und TA) soll hier kurz erläutert werden.

Die Daten des Text Arrays werden in einem B^+-Baum abgelegt, der nach der Position des Textabschnitts im Dokument sortiert ist. Die Position ist hierbei bytegenau. Um trotzdem Änderungsoperationen auf dem TA effizient unterstützen zu können, werden in den Knoten des B^+-Baums nur relative Positionsangaben gespeichert. Wenn ein Textabschnitt neu eingefügt wird, dann müssen nur die relativen Positionen in den Elternknoten maximal hoch bis zur Wurzel angepasst werden. Dies verursacht im Vergleich zu einem kompletten, neuen Indexaufbau nur minimale Kosten.

Der AST^P ist ein Index, der im Wesentlichen alle DOM-Knotenarten bis auf Textknoten unterstützt. Anstelle der Textknoten werden Verweise auf Positionen im TA verwendet. Wie beim Text Arrray werden auch hier in den Baumknoten nur relative Positionen angegeben, um gegen Verschiebungen bei Einfüge- oder Änderungsoperationen im TA möglichst immun zu sein.

Die Speicherung der Knoten von AST^P erfolgt in Blöcken auf dem Externspeicher. Hierbei werden immer alle Kinder von einem Elternknoten zu einer Gruppe von Knoten zusammengefasst. In der Regel finden mehrere solche Gruppen in einem Block Platz. Wird eine Gruppe zu groß für einen Block, so wird sie aufgeteilt und mit Vorwärts- und Rückwärtsverkettung über mehrere Blöcke verteilt gespeichert.

Diese Art der Abbildung auf Externspeicherblöcke wurde dem Papier eines japanischen Projekts [KYU01] entnommen, welches sich hauptsächlich mit der Indexierung von XML befasst. Dort werden auch Split- und Mergealgoritmen angegeben, um dynamische, baumverändernde Operationen effizient zu unterstützen. Die Algorithmen basieren auf den Ideen des Bulkloadings, wobei immer möglichst große Teilbäume abgespalten werden. Diese Technik ist allerdings beim Normalfall eines Splits problematisch. Bei den meisten Splits soll ein nur minimal zu großer Baum aufgeteilt werden. Durch den vorgeschlagenen Splitalgorithmus entstehen häufig ein großer und ein kleiner Teilbaum, was für die Leistung der Speicherungsstruktur insgesamt nicht förderlich ist.

Bei allen in diesem Abschnitt genannten Papieren fällt leider negativ auf, dass lediglich neue Strukturen vorgeschlagen werden, diese jedoch keinen praktischen Leistungsbestimmungen unterzogen werden. Theoretische Aussagen zu Leistungsfragen sind ebenso nicht vorhanden. Ein direkter Vergleich mit anderen Verfahren aus der Literatur ist somit nicht möglich.

5.6.4 Efficient Native XML Storage System (ENAXS)

Auch ENAXS [WNL03] trennt bei der Speicherung die Struktur vom Inhalt. Interne Baumknoten werden getrennt von externen Textknoten in unterschiedlichen Bereichen abgelegt.

An dieser Stelle soll nur den strukturelle Teil genauer betrachtet werden. In einem Block auf dem Externspeicher werden hierbei mehrere interne Knoten gespeichert. Wenn ein neuer Knoten eingefügt wird, so geschieht dies immer im letzten Block. Was dies genau bedeutet, ist der Veröffentlichung nicht zu entnehmen. Wurden zwischenzeitlich einige Knoten gelöscht, so stellt sich die

Frage, ob dann immer noch in den physisch letzten Block eingefügt werden soll. Von einer Freispeicherverwaltung ist in dem Papier jedenfalls keine Rede.

Generell richtet sich die Position eines internen Knotens nicht nach der Position im Dokument, sondern nur nach der Einfügereihenfolge. Somit implementiert ENAXS einen Element Split, bei dem im Unterschied zum Element Split in der Natix Struktur die Textknoten in einer zweiten Struktur abgelegt werden. Warum das Konzept der zwei Strukturen überhaupt von Vorteil sein soll, wird ebenfalls nicht erläutert. Meiner Meinung nach würde die Trennung nur dann einen Sinn ergeben, wenn eine der Knotenarten immer dieselbe Anzahl von Bytes belegen würde und hierdurch eine effizientere Struktur als ein Recordmanager benutzt werden könnte. Dies ist jedoch offensichtlich nicht der Fall.

Ein wesentlicher Punkt bei der Entwicklung von ENAXS war die Anbindung von Indexstrukturen. Es wird eine neue Indexart vorgeschlagen, die jeden Pfad auf einen Hashwert abbildet. Die Hashwerte werden in einem B^+-Baum verwaltet und verweisen auf Knotenreferenzmengen, die wiederum auf die eigentlichen Daten verweisen. Neben dieser Struktur existiert noch ein Schemabaum, der ebenfalls zur Beschleunigung der XPath-Auswertung eingesetzt wird.

Bei den Experimenten werden leider keine Vergleiche zu einfachen nativen Speicherungsstrukturen gezogen. Es werden lediglich Parameter der eigenen Struktur optimiert, was letztlich zu einer verbesserten Speicherplatzausnutzung führt. Weiterhin wird festgestellt, dass die eingeführten Indexstrukturen teilweise extreme Leistungsgewinne ermöglichen. An dieser Stelle werden jedoch nur die Ergebnisse für selbst gewählte XPath-Anfragen gezeigt und kein vorhandener Benchmark benutzt. Leider ist auch kein Experiment vorhanden, das den Aufwand für die Aktualisierung der Indexe nach baumverändernden Operationen ermittelt.

5.7 Zusammenfassung und Ausblick

In diesem fünften Kapitel wurde zunächst der Begriff des nativen XML Datenbanksystems eingeführt. Solch ein System muss eine Speicherungsstruktur verwenden, welche die Struktur der Dokumente bei der Speicherung berücksich-

tigt. Danach wurde eine entsprechende Speicherungsstruktur, die des nativen XML Datenbanksystems Natix, vorgestellt.

Auf dieser Basis erfolgte die Implementierung einer nativen Speicherungsstruktur für XML in XXL. Es wurden neue Splitalgorithmen vorgeschlagen und diese mit den aus der Literatur stammenden Algorithmen theoretisch und praktisch verglichen. Der Onecut Split mit Scaffold erwies sich bei ausführlichen Tests als überlegenes Verfahren. Nicht vergessen werden darf, dass dieser Splitalgorithmus als einziger auch ohne Recordmanager betrieben werden kann, wodurch die Anzahl der Externspeicherzugriffe weiter gesenkt werden kann.

Bei Anfragen auf nativem Speicher zeigte sich, dass Dokumente, die mit Bulkloading aufgebaut wurden, eine deutlich bessere Anfrageleistung bieten. Dies war in der Literatur bislang nur vermutet, nicht aber nachgewiesen worden.

Anschließend wurden Indexstrukturen für nativen Speicher diskutiert. Es wurde die Problematik von referenzierenden Indexstrukturen beschrieben und Lösungsmöglichkeiten aufgezeigt. Weiterhin wurde in den nativen Speicher von XXL ein nicht referenzierender Signaturindex integriert. Er verwendet Signaturen als Aggregate an jedem Knoten und belegt auf dem Externspeicher sehr wenig Speicherplatz. Bei den Tests stellte sich heraus, dass hierdurch die Anfrageleistung bei XPath-Anfragen um mehr als 25% gesteigert werden kann.

In der Praxis stellen sich eine Reihe weiterer Fragen, die in diesem Kapitel unbeantwortet bleiben mussten. So ist es vorstellbar, dass größere, recordübergreifende Teilbäume in einem Dokument nach vielen erfolgten Splits wieder reoptimiert werden. Wie sich gezeigt hat, wird die Qualität der Speicherungsstruktur im Laufe der Zeit immer schlechter. Eine Reoptimierung kann die Anfrageleistung des Systems deutlich erhöhen und sollte deshalb immer dann erfolgen, wenn wenig Last im System vorhanden ist.

Um eine lohnende Stelle für eine Reoptimierung schnell finden zu können, besteht die Möglichkeit, neue Aggregate einzuführen. Hierdurch kann in jedem Knoten die Anzahl an Splits gespeichert werden, die unterhalb des Knotens aufgetreten sind. Bei einer Reoptimierung wird mit Hilfe dieses Aggregats dann nach einer Stelle gesucht, wo im Verhältnis zur Teilbaumgröße relativ viele Splits erfolgt sind.

Die eigentliche Reoptimierung läuft dann wie folgt ab. Für den betroffenen Teilbaum, der nicht notwendiger Weise bis zu den Blättern des XML Baums gehen muss, werden Ereignisse generiert und diese für ein Bulkloading verwendet. Anschließend erfolgen nur noch das Einhängen im Dokument und das Löschen der betroffenen Records des ursprünglichen Teilbaums[16].

Neben der dynamischen Reoptimierung von Teilbäumen sollen in Zukunft Kompressionsverfahren in der XML Speicherungsschicht genauer untersucht werden. Die widerstrebenden Ziele wurden in Kapitel 5.2.3 angesprochen.

Künftige Arbeiten sollten sich darüber hinaus mit der Integration von referenzierenden Indexstrukturen im Framework von XXL befassen. Hier gibt es einige spannende Fragen, die es zu beantworten gilt, beispielsweise, bei welchen Änderungsraten sich diese Indexe überhaupt auszahlen. Weiterhin sollen die einzelnen Splitalgorithmen an den Signaturindex angepasst werden. Hier ist es von Interesse, bei welchen Verhältnissen von baumverändernden Operationen zu Suchoperationen der Index einen Vorteil bringt.

Das momentane Grundübel bei nativem XML Speicher ist allerdings die schlechte Situation bezüglich der Evaluierung der vorgeschlagenen Strukturen. An dieser Stelle empfehle ich, dass grundsätzlich immer die Leistungswerte des Element Splits als Referenz angegeben werden sollen. Das Element Split Verfahren ist zum einen einfach zu implementieren und zum anderen schon in einigen Systemen vorhanden. Darüber hinaus existieren einige open-source Implementierungen des Verfahrens, so z. B. auch in XXL.

Im folgenden sechsten Kapitel werden einige Vergleiche zwischen den beschriebenen nativen Verfahren und relationalen Speicherungsstrukturen für XML durchgeführt. Solche Vergleiche sind bislang nur auf der Basis von Systemtests erfolgt. Bei solchen Tests besteht jedoch keine ausreichende Kontrolle über die verwendeten Ressourcen. Daher lassen diese Tests keine Aussagen über die Leistung von Verfahren zu.

[16] Das Löschen der Records kann zeitverzögert erfolgen.

Kapitel 6

Relationale Speicherung von XML

In diesem Kapitel verlassen wir zunächst den Bereich nativer XML Datenbanksysteme und wenden uns relationalen XML Datenbanksystemen zu. In der Literatur sind weit mehr Ansätze zu finden, die eine Speicherung in Relationen durchführen, als Ansätze, die XML in speziellen nativen Strukturen ablegen [DFS99,SKWW01,Suc01,AS02,CS03,LLG03]. Die Ursache hierfür mag sein, dass viele Forscher vorhandene Systeme nutzen wollen und nicht unbedingt die Notwendigkeit für aufwendige Neuentwicklungen sehen [STZ+99].

Vergleiche zwischen relationalen und nativen XML Systemen sind in der Literatur bisher rar. Es existieren bislang nur Vergleiche, in denen die Laufzeit von realen Systemen für die Bewältigung einer vorgegebenen Aufgabe gemessen wurde. Als Beispiel hierfür ist die Arbeit in [BSM+03] zu nennen, wo XML Benchmarks auf der Basis von nativen relationalen XML Datenbanksystemen implementiert wurden und ein Vergleich der Laufzeiten erfolgte.

Dies ist jedoch eine Art von Vergleich, die keine Aussagen über die Effizienz der zugrundeliegenden Strukturen zulässt. Es besteht bei den Messungen weder eine Kontrolle über die tatsächlich verwendete Hauptspeichergröße, noch eine Kontrolle über die einzelnen Komponenten der Systeme. Wenn beispielsweise der Recordmanager von einem System nicht ausgereift ist, leidet die Performanz der XML Speicherungsstruktur eventuell so stark darunter, dass die Gesamtleistung trotz eventuell besserer XML Verarbeitung niedriger ist. Wie bereits in

Kapitel 2.2 beschrieben, treten weitere Effekte auf, die in der Wahl der Implementierungssprache und in den verwendeten Compilern begründet sind. Somit kann aus einer Zeitmessung, bei der System A um 20% schneller ist als System B, nicht geschlossen werden, dass System A die besseren internen Strukturen für die durchgeführte Aufgabe besitzt.

Es ist also dringend erforderlich, dass Tests von Speicherungsstrukturen innerhalb eines einzigen Systems erfolgen. Hierbei müssen dieselben grundlegenden Komponenten Verwendung finden. Dies ist nun erstmals durch Benutzung von XXL und seiner neu entwickelten Komponenten möglich.

Das nun folgende Kapitel beschreibt zunächst drei sehr verschiedenartige Ansätze, die in der Literatur für die Speicherung von XML in relationalen Datenbanksystemen vorgeschlagen wurden [TDCZ02]. Daran anschließend wird auf deren Implementierung im Datenbank-Framework von XXL genauer eingegangen. Die Anfrageleistung der Verfahren wird dann durch die Ausführung von Anfragen des XMark-Benchmarks gemessen und gegenübergestellt. In diesem Zusammenhang erfolgt ein Vergleich mit den Leistungswerten des nativen XML Speichers von XXL. Abschließend enthält das Kapitel noch eine kurze Zusammenfassung und einen Ausblick.

6.1 Speicherung von XML in Relationen

Es wurde bereits angesprochen, dass in der Literatur eine Reihe Vorschlägen von Speicherungsverfahren zu finden sind, welche ihre Daten in Relationen von herkömmlichen Datenbanksystemen ablegen. [TDCZ02] beschreibt drei Verfahren dieser Art, die typisch für eine relationale Speicherung von XML sind. Diese Verfahren sollen im Folgenden vorgestellt werden.

6.1.1 Speicherung als Ganzes

Das erste Verfahren verwendet nur eine einzige Relation `Dokumente(Dokumentname String, Dokumentdaten CLOB)`. Die XML Dokumente werden hier also als Ganzes in einem CLOB abgelegt. Bei den kleinsten Änderungen muss

das komplette Dokument geladen, geändert und neu geschrieben werden, was einen extremen Aufwand bedeutet. Bei Anfragen muss entweder ein DOM-Baum im Hauptspeicher aufgebaut werden oder das Dokument mit einem entsprechenden XPath für SAX untersucht werden.

6.1.2 Der DTD Ansatz

Der DTD Ansatz geht davon aus, dass nur Dokumente gespeichert werden, die einer Grammatik in Form einer DTD gehorchen. Ist keine DTD vorgegeben, so muss diese zuerst ermittelt werden. Hierzu ist vor dem Einfügen des ersten Dokuments ein SAX Durchlauf über alle zu speichernden XML Dokumente nötig[1].

Anschließend wird für jede aus der DTD ableitbare Knotenart eine eigene Relation angelegt. Unter Knotenart ist hier zu verstehen, dass alle in ihr enthaltenen Knoten die gleichen Namen und die gleichen Wurzelpfade[2] besitzen. Die Namen der verwendeten Relationen spielen keine Rolle. Sie werden auf ein gewähltes Präfix, gefolgt von einem numerischen Identifikator gesetzt, so dass es keine zwei gleich benannten Relationen gibt. Als Attribute enthält jede der Relationen einen Dokumentidentifikator `dokId` (Integer), einen innerhalb eines Dokuments eindeutigen Knotenidentifikator `kId` (Integer), sowie einen Verweis auf den Elternknoten `parentId` (ebenfalls Integer). Zusätzlich sind in jeder Relation ein Attribut pro XML-Attribut vorhanden sowie ein Stringfeld, in dem die zum Knoten gehörenden Texte abgelegt werden.

Eine Relation mit Namen Präfix+"Relationen" stellt die Zuordnung von Knotenarten zu den entsprechenden Relationen her. Diese Verweisrelation besitzt Attribute für den Knotennamen (String), den Wurzelpfad (String) und den Identifikator der Relation (Integer). Schließlich existiert noch eine weitere Relation mit dem Namen Präfix+"Dokumente", die den Dokumenten eindeutige Identifikatoren (Integer) zuweist.

[1] Oder: Es muss ein Dokument vorhanden sein, das alle verschiedenen Regeln der Grammatik einsetzt. Dann reicht es, nur aus diesem Dokument eine Grammatik zu extrahieren.

[2] Der Wurzelpfad ist die Folge von Knotennamen auf dem Weg von der Wurzel zum betrachteten Knoten.

Probleme gibt es bei dieser Art der Speicherung mit dem so genannten Mixed-Content. Wenn in einer DTD ein Knoten eine beliebige Folge von Texten und anderen Knoten enthalten darf, so sind die mit diesem Ansatz gespeicherten Dokumente nicht 1:1 rekonstruierbar. Wenn beispielsweise mehrere Texteinträge zu einem Knoten vorhanden sind, so geht bei der Wiederherstellung des Dokuments die Information verloren, welcher Texteintrag wo zugeordnet werden muss. Ein Beispiel: Für das Dokument

```
<a>Hallo<b/>Welt</a>
```

würde der komplette Textanteil „Hallo Welt“ in einem Tupel der a-Relation abgelegt. Bei der Wiederherstellung würde hierdurch das Dokument

```
<a>HalloWelt<b/></a>
```

entstehen. Eine Lösung dieses Problems würde zu einem deutlich schlechteren Schema führen. Bei datenzentrischen Dokumenten ist Mixed-Contents auch eher unüblich, weshalb dieser Fall in der angegebenen Quelle nicht weiter betrachtet wurde.

6.1.3 Der Kantenansatz

Die dritte relationale Speicherungsvariante bildet die Kanten der Baumstruktur von XML-Dokumenten in eine Relation `Kanten` ab. Das Schema der Relation sieht wie folgt aus: `Kanten(KnotenId Integer, ElternId Integer, Knotenname String, Reihenfolgeposition Integer, Text String)`. Die Reihenfolgeposition bestimmt eindeutig, an welcher Stelle ein Knoten unterhalb seines Elternknotens steht[3]. Eine „0“ als Reihenfolgeposition bedeutet, dass es sich bei dem Eintrag um ein Attribut handelt[4].

[3] Mixed-Content ist hierdurch Problemlos möglich.

[4] Dann entspricht der Knotenname dem Attributnamen und der Text dem Attributwert. Auf eine Reihenfolge innerhalb von Attributen wird bei XML meistens verzichtet, da nicht die Reihenfolgeposition, sondern der Attributname zur Identifizierung der Information dient.

Zusätzlich zu dieser Relation existiert eine weitere Relation, welche ähnlich wie beim DTD-Ansatz auf die Wurzeleinträge der einzelnen Dokumente verweist.

6.1.4 Zusammenfassung

Die drei beschriebenen Modelle legen XML Dokumente in Relationen ab. Der DTD Ansatz und der Kantenansatz zerlegen dabei die Dokumente in kleine Bestandteile, die über viele Tupel verstreut abgelegt werden.

Für die Effizienz der Anfrageverarbeitung ist wichtig, welche Indexstrukturen benutzt werden können. Generell ist es sinnvoll, einen Index für die Lokalisierung der Dokumente bzw. deren Wurzeln anzulegen. Der Textansatz kommt damit bereits aus und braucht keine weiteren Indexstrukturen.

Beim DTD Ansatz hängt viel davon ab, wie viele Relationen erzeugt werden und wie voll diese tatsächlich sind. Für gut gefüllte Relationen macht der eine oder andere Index sicherlich Sinn. Eine Indexierung der Verweisrelation ist nur auf dem Wurzelpfadattribut sinnvoll. Hierfür kann ein Präfix-B^+-Baum eingesetzt werden, falls im verwendeten Datenbanksystem so eine Struktur zur Verfügung steht.

Beim Kantenansatz ist ein Index über die Elternidentifikatoren essentiell für die Leistung, weil Anfragen der folgenden Art häufig auftreten: Gebe mir alle Kinderknoten des Knotens mit der Nummer x. Somit müssen in der Kantenrelation alle Tupel gefunden werden, deren Elternidentifikator x ist, wozu aus Effizienzgründen ein Index wichtig ist.

6.2 Beschreibung der Implementierung in XXL

Bei der Implementierung in XXL wurde das bereits in Kapitel 3.2.4 beschriebene Datenbank-Framework eingesetzt. Die Relationen werden hierbei innerhalb eines Recordmanagers abgelegt, der seine Daten in einem blockbasierten Container speichert. In unserem Fall werden für alle Relationen derselbe Container benutzt, weil dies die Zählung der Externspeicherzugriffe vereinfacht und sonst

nicht nachteilig ist. Die Tupel werden mittels eines `Converter`s in Records umgewandelt und im Recordmanager der entsprechenden Relation gespeichert. Die `TupelConverter` sorgt dafür, dass alle String-Felder nur so viel Platz auf dem Externspeicher belegen, wie es aufgrund der Stringlänge nötig ist. Es wird kein Platz mit Leerzeichen verschwendet. Demzufolge gibt es keine feste Zuordnung von Attributwerten zu Positionen in den Records.

Beim Textansatz wird das CLOB im Unterschied zur obigen Beschreibung auf mehrere aufeinander folgende Records verteilt. Grund hierfür ist, dass der Recordmanager in XXL keine größeren Records als $maxRecordSize = B - 9$ unterstützt.

Das Datenbank-Framework erlaubt die Definition von B^+-Bäumen als Indexstrukturen. Für jede Relation können beliebig viele Indexe angelegt werden, die bei Einfüge-, Änderungs- und Löschoperationen von Tupeln automatisch aktualisiert werden.

Zur Beantwortung von XPath-Anfragen werden auf den drei Strukturen verschiedene Techniken eingesetzt. Beim Textansatz wird das Dokument zusammengesetzt und im Hauptspeicher als DOM-Baum aufgebaut. Die Auswertung der XPath-Anfragen übernimmt Apache Xalan.

Für die anderen beiden Strukturen wurden einfache, selbst entwickelte XPath Prozessoren aufgesetzt. Diese machen Gebrauch von der physischen, relationalen Operatoralgebra von XXL. Im Fall des Kantenansatzes wird exzessiv von dem bereits beschriebenen Index über dem Elternidentifikator Gebrauch gemacht. Für jede direkte Eltern-Kind-Relation wird logisch gesehen ein Join ausgeführt, der als Index-Selfjoin physisch realisiert wurde.

Beim DTD-Ansatz liegt die Hauptlast auf dem Lesen der Verweisrelation. Ein Index wurde aus zwei Gründen hierfür nicht verwendet. Zum einen hilft ein Index bei XPath-Anfragen mit einer `descendant-or-self` Achse nicht weiter. Hier muss sowieso die komplette Verweisrelation gelesen werden. Zum anderen ist die Verweisrelation nur wenige Blöcke groß. Das sequentielle Lesen der Verweisrelation ist bei den betrachteten Dokumenten effizienter als die Benutzung von Indexen, die wahlfreie Zugriffe zur Folge hätten.

6.3 Evaluierung

Eine Domäne von nativem Speicher sind Navigationsoperationen. Bei diesen Operationen ist der Vorteil von teilbaumbasierten, nativen Strukturen unbestritten. Auf entsprechende Messungen wird an dieser Stelle daher verzichtet.

Die Hoffnung der relationalen XML Systeme ist, dass sich das set-at-a-time Prinzip gegenüber nativen XML Strukturen auszahlt. Gerade beim DTD-Ansatz ist dies eine Erwartung, weil nach dem Auffinden eines Eintrags in der Verweisrelation alle Ergebnisse durch das sequentielle Lesen der entsprechenden Knotenrelation auf einmal generiert werden können. Dies ist ein deutlicher Vorteil bei der Ausführung von XPath-Anfragen.

Zum Zweck der Leistungsmessung wurde erneut das XMark1-Dokument verwendet und von den XMark-Anfragen diejenigen ausgewählt, die auf relationalem Speicher ohne Navigation auskommen. Weiterhin wurden keine Anfragen verwendet, die sich auf eine Position relativ zu einer Knotenart unterhalb des Elternknotens beziehen (Beispiel: `/a/b[7]`). Diese Art Anfrage wird beim DTD-Ansatz im Fall von Mixed-Content, wie er im XMark-Dokument vorkommt, generell nicht unterstützt.

Auch beim Kantenansatz ist übrigens die Berechnung entsprechender Positionsnummern sehr aufwendig. Solche Nummern beziehen sich ausschließlich auf gleichnamige Geschwisterknoten und sind somit nicht identisch mit den Positionen, die in der Kantenrelation sowieso schon gespeichert werden. Für jede Positionsbestimmung muss daher eine zusätzliche Indexanfrage ausgeführt werden, wodurch enorme Kosten entstehen. Bei einzelnen Tests zeigte sich nativer Speicher bei diesen Anfragen deutlich überlegen.

In der Evaluierung werden nun die drei relationalen Verfahren mit Bulkloading (mit Verwendung von Signaturen, Signaturlänge 64 Bit, Signaturgewicht 4 Bit) verglichen. Der Aufbauprozess der vier Strukturen nimmt relativ wenig Zeit in Anspruch. Hier treten wenige wahlfreie Zugriffe auf, und die Unterschiede sind gering (daher nicht in einem Diagramm dargestellt).

Als nächstes stellt sich die Frage, wie viel Platz die Verfahren auf dem Externspeicher belegen. Dies zeigt Abbildung 6.1 in Abhängigkeit zur Blockgröße. Für Bulkloading, den Text- und den Kantenansatz zeigt sich keine deutliche Abhängigkeit von der Blockgröße. Beim DTD-Ansatz steigt der belegte Speicherplatz demgegenüber nahezu linear mit der Blockgröße an. Dies liegt daran, dass der DTD-Ansatz beim verwendeten Dokument sehr viele Relationen erzeugt (>250). Jede Relation muss in XXL allerdings mindestens einen Externspeicherblock belegen, wodurch viel Speicherplatz verschwendet wird. Blockgrößen von mehr als 512 Bytes scheiden daher bei diesem Ansatz aus Speicherplatzgründen aus.

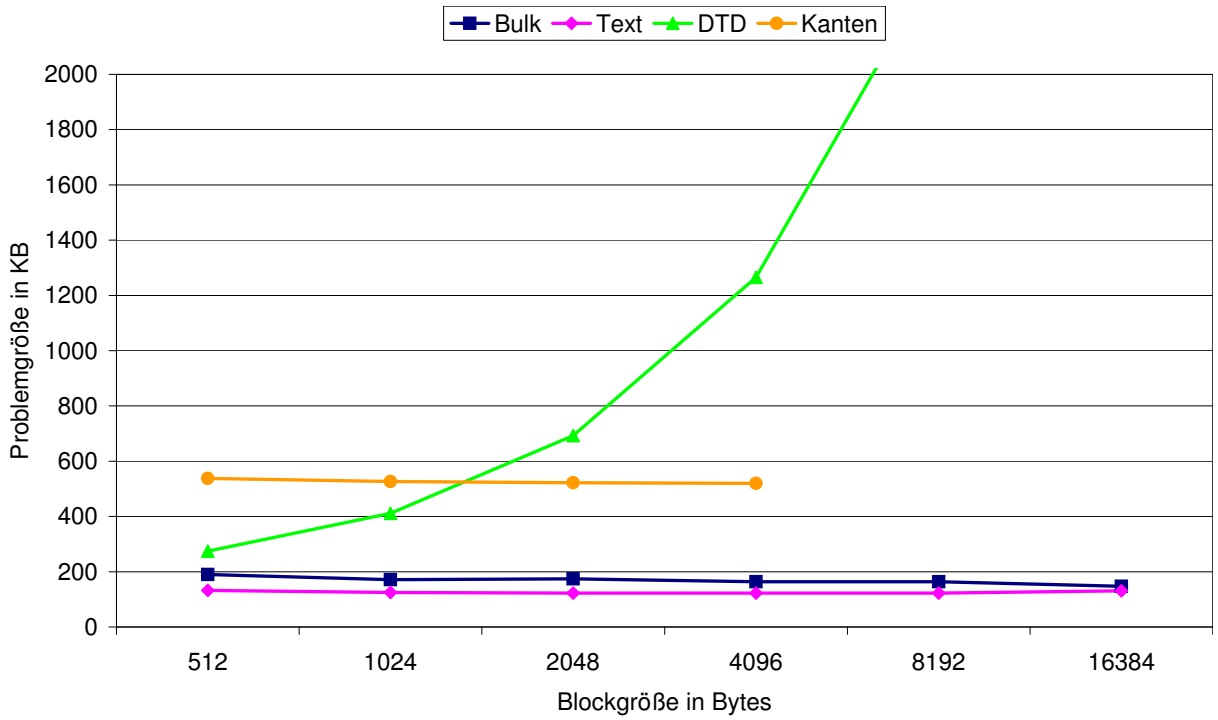


Abbildung 6.1: Problemgröße in KB für verschiedene Speicherungsverfahren und Blockgrößen. P=16384.

Der Kantenansatz weist generell eine schlechte, aber gleichbleibende Problemgröße auf. Die Messungen mit Blockgrößen von 8KB bzw. 16KB konnten bei ihm nicht durchgeführt werden, da die verwendete Puffergröße von 16KB für die `fix`-Operationen innerhalb des B^+-Baums nicht ausreichend war. Bulkloading und der Textansatz besitzen erwartungsgemäß die geringsten Problemgrößen von den vier Verfahren.

Als Zweites betrachten wir nun die Anzahl der gewichteten Zugriffe für die Ausführung der XPath-Anfragen. Eine Gewichtung von sequentiellen und wahlfreien Zugriffen ist hier nötig, weil der Textansatz extrem viele sequentielle Externspeicherzugriffe erzeugt, wohingegen bei den anderen Verfahren die Kosten für die wahlfreien Zugriffe dominieren. Im Folgenden wurde eine Gewichtung von 1:20 gewählt, also ein wahlfreier Zugriff besitzt dieselben Kosten wie 20 sequentielle Zugriffe.

Bei Betrachtung der Anfrageleistung (Abbildung 6.2) stellt man fest, dass sich bei allen Verfahren eine größere Blockgröße positiv auf die Anfrageleistung auswirkt. Bei kleinen Blockgrößen ist der DTD-Ansatz am schnellsten. Ab einer Blockgröße von 4096 Bytes übernimmt der Textansatz die Spitzenposition.

Der Kantenansatz ist auch aus Sicht der Anfrageleistung die schlechteste Wahl, und zwar unabhängig von der Blockgröße. Er zeigt ein sehr ähnliches Verhalten wie der Element Split bei nativem Speicher, wozu es in der Struktur auch einige Parallelen gibt.

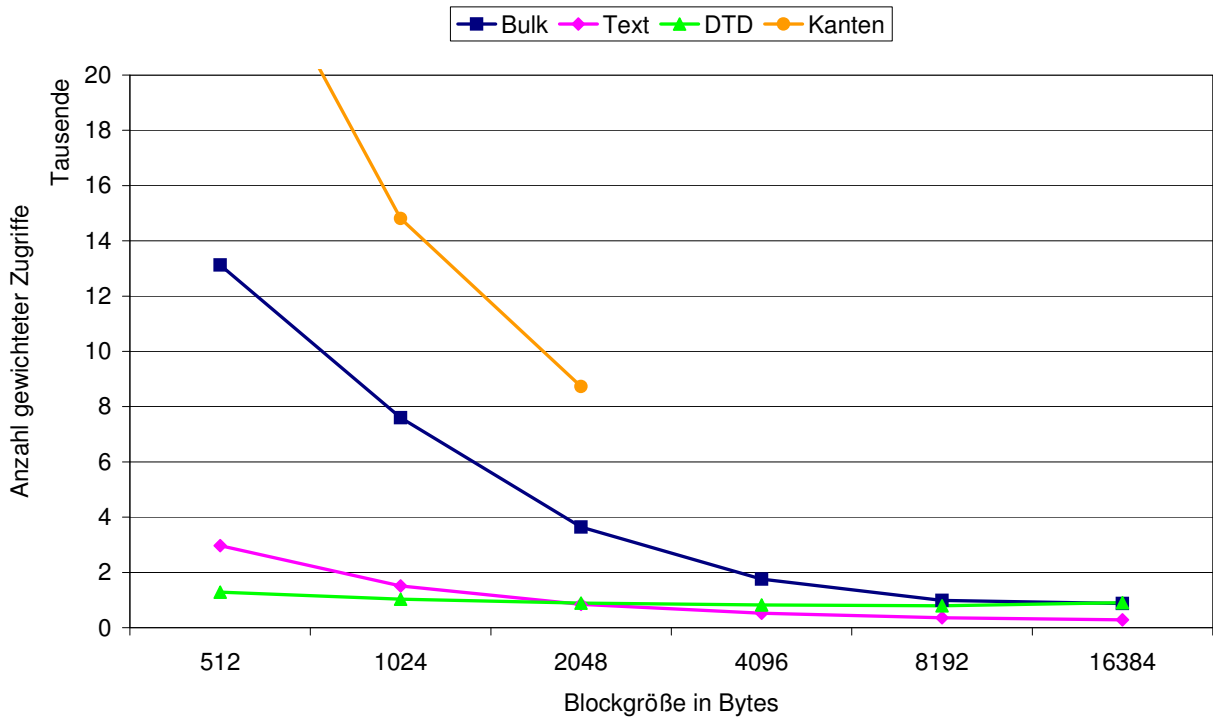


Abbildung 6.2: Anzahl gewichteter Zugriffe für XPath-Anfragen bei verschiedenen Speicherungsverfahren und Blockgrößen. P=16384.

Wenn man Anfragen auf nicht zu großen, selten veränderten Dokumenten hat, so ist die Speicherung als Ganzes keine schlechte Wahl. Treten jedoch Navigations- oder Änderungsoperationen auf, so ist der Ansatz nicht verwendbar.

In diesem Fall bleiben noch die native XML Speicherungsstruktur und der DTD-Ansatz als Alternativen übrig. Der DTD-Ansatz kann aus Gründen der Speicherplatzbelegung nur mit der kleinsten möglichen Blockgröße von 512 Bytes betrieben werden. Bei dieser Blockgröße erledigt er die Anfragen mit 1290 wahlfreien Zugriffen. Bulkloading kann demgegenüber auch gut bei 16 KB großen Blöcken betrieben werden. Hier kommen die Anfragen mit 878 wahlfreien Zugriffen aus, was knapp 47 Prozent weniger ist als beim DTD-Ansatz.

Zuletzt betrachten wir noch die CPU-Zeit für die Ausführung der Anfragen. Sie ist in Abbildung 6.3 dargestellt. Durch das wiederholte Aufbauen des DOM-Baums beim Textansatz ist dieser hier mit Abstand das langsamste Verfahren. In dieser Disziplin sind Bulkloading und der DTD-Ansatz in etwa gleichauf.

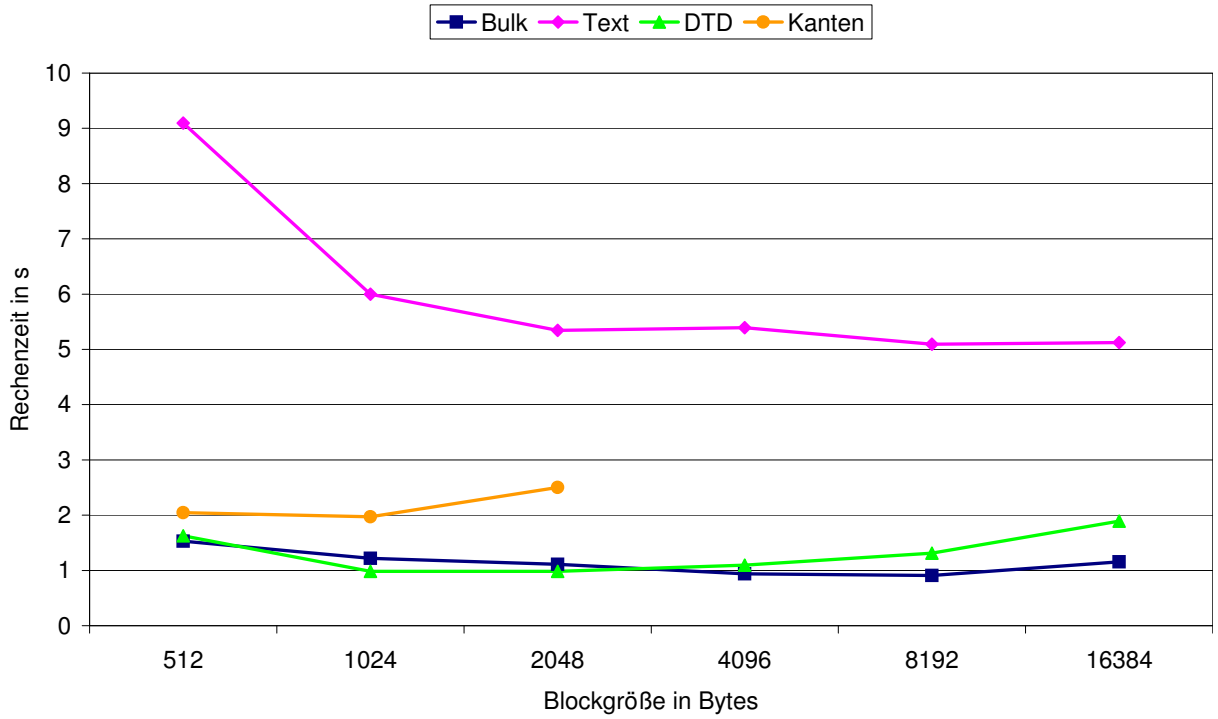


Abbildung 6.3: Rechenzeit für die XPath-Anfragen der verschiedenen Speicherungsverfahren im Vergleich zur Blockgröße. P=16384.

6.4 Zusammenfassung und Ausblick

In diesem Kapitel haben wir drei Verfahren für die Speicherung von XML in relationalen Datenbanksystemen betrachtet. Diese Verfahren wurden im Datenbank-Framework von XXL implementiert und mit dem nativen Speicher aus Kapitel 5 verglichen.

Hierbei stellte sich heraus, dass der native Speicher bei den XPath-Anfragen gute Leistungen zeigt. Die relationalen Verfahren besitzen alle gewisse Schwachstellen, weshalb der native Speicher insgesamt als Sieger aus den Experimente hervorgeht.

Der Kantenansatz besitzt eine deutlich größere Problemgröße und zusätzlich keine besseren Leistungsmerkmale als der native Speicher. Der Textansatz ist nur bei im Verhältnis zur Anzahl der Anfragen wenigen Änderungsoperationen sinnvoll einsetzbar. Dann zeigt er jedoch teilweise ein gutes Leistungsvermögen. Eine interessante Erweiterung dieses Ansatzes wäre es, wenn zusätzlich zu den Dokumenten noch jeweils ein Strukturindex in einem CLOB abgelegt würde. Wenn die Dokumente sich nicht häufig verändern, hätte dies keine großen Nachteile bei deutlichen Vorteilen in der Anfrageverarbeitung.

Beim DTD-Ansatz sind die erzeugten Strukturen verhältnismäßig groß. Weiterhin gibt es noch ein grundsätzliches Problem. Wenn bei echt semi-strukturierten Daten neue Knoten eingefügt werden, die zusätzliche Attribute enthalten, müssen alle Tupel der betroffenen Relation angepasst werden. Somit darf sich zur Laufzeit das Schema der betrachteten Dokumente auf keinen Fall ändern. Es lässt sich sagen, dass der DTD-Ansatz für datenzentrisches XML sehr gut geeignet ist, jedoch bei echt semi-strukturierten Daten deutliche Schwächen besitzt.

Die eigentliche Stärke des nativen Speichers sind jedoch navigierende Anfragen. An diesen Stellen kann keine der relationalen Speicherungsstrukturen auch nur annährend mithalten.

Kapitel 7

Bedarfsgesteuerte XML Ströme

Das Internet ist die größte Datenansammlung unserer Zeit. In ihm sind Informationen zu fast jedem nur vorstellbaren Thema enthalten. Es ist in vielen Bereichen ein Spiegel für den aktuellen Wissensstand der Menschheit. Im Alltag ist das Extrahieren von Informationen aus dem Internet inzwischen zur Normalität geworden. Da viele Anwender einen nicht unbeträchtlichen Teil ihrer Arbeitszeit für die Informationssuche aufwenden, sind automatische und die Effizienz steigernde Suchmethoden von großem Interesse.

Meistens ist eine einzelne Information für sich genommen nicht viel Wert. Erst die Verbindung mit weiteren Informationen erzeugt Zusammenhänge, die den Anwender wirklich interessieren. Somit ist die Notwendigkeit gegeben, Quellen aus dem Internet in eine flexible und ausdrucksstarke, aber dennoch für den Informationssuchenden einfach anwendbare Anfrageverarbeitung einzubauen.

Das klassische Beispiel für solch eine Anfrage lautet etwa wie folgt: „Finde mir einen günstigen Flug, vorher eine passende Bahnverbindung zum Flughafen (ohne zu viele Zugwechsel) und am Zielort in der näheren Umgebung des Konferenzzentrums ein möglichst günstiges Hotel. Suche mir zusätzlich ein zu meinen Vorlieben passendes Restaurant für den ersten Abend, und stelle mir außerdem eine Liste mit kulturellen Veranstaltungen während meines Aufenthalts zusammen.“

Wenn man für eine solche Anfrage keinen Dienstleister beschäftigt, so ist die Suche entsprechender Angebote im Internet extrem zeitaufwendig. Besäße man eine strukturierte Datenbank, die alle benötigten Informationen enthält, wäre es möglich, ein SQL-Skript zur Beantwortung der Anfrage zu schreiben. Dies wäre sicherlich keine einfache Aufgabe, da Präferenzen, Lokationen und implizites Wissen hier eine Rolle spielen, allerdings ist das Problem prinzipiell lösbar.

Im Internet geht es jedoch zuerst einmal darum, überhaupt alle Seiten mit passenden Informationen zu lokalisieren. Aus den gefundenen Quellen müssen anschließend die interessanten Informationen extrahiert werden[1], die dann in der Anfrageverarbeitung miteinander in Beziehung gesetzt werden.

In diesem Kapitel geht es um die Grundlagen von Systemen zur Verarbeitung solcher Anfragen. Gleichzeitig kann dieselbe Anfrageverarbeitung auch für den nativen XML Speicher eingesetzt werden, da das Datenformat relativ ähnlich ist.

Als Basis meines Ansatzes dienen Ströme von XML Dokumenten. Woher die XML Dokumente stammen, spielt keine Rolle. Sie können aus einem nativen XML Datenbanksystem stammen, lokal als Dateien vorliegen, aus vorhandenen Quellen generiert werden oder auch aus dem Internet stammen. Letztere werden im System i. d. R. nicht dauerhaft gespeichert, sondern zum Zeitpunkt der Anfrage lokalisiert und zum Anfragesystem übertragen. Hierdurch wird die Aktualität der gewonnenen Informationen sichergestellt. Mechanismen des Cachings können an Stellen, an denen die Aktualität nicht so wichtig ist, zur Leistungsverbesserung eingesetzt werden[2].

Die wesentliche Idee bei den im Folgenden dargestellten Arbeiten ist die Kopplung von verschiedenen Datenquellen mit einer flexible Anfrageverarbeitung, wie sie von XXL zur Verfügung gestellt wird. Hierzu wurde ein Framework entwickelt, dessen grundlegender Aufbau in Kapitel 7.1 vorgestellt wird. Anschließend werden die Anbindung von Webservices diskutiert (Kapitel 7.2) und

[1] Hierbei ist es unter anderem wichtig, zeitliche und räumliche Bezüge zu erkennen. Dies ist bereits ein schwerwiegendes Problem für automatische Verfahren.

[2] Es gibt hier einen Tradeoff zwischen Performanz und Aktualität.

einige Beispielszenarien vorgestellt (Kapitel 7.3). Kapitel 7.4 beschreibt Arbeiten aus der Literatur, die in diesem Zusammenhang von besonderer Bedeutung sind. Abschließend erfolgt eine kurze Zusammenfassung und ein Ausblick auf künftige Arbeiten.

7.1 Framework

Die nun beschriebene Anfrageverarbeitung auf XML Strömen wurde in XXL im Paket `xxl.core.xml.operators` implementiert. Die Verarbeitungsoperatoren setzen auf der Cursor-Algebra von XXL auf (siehe Kapitel 3.1.2), d. h. die Verarbeitung erfolgt bedarfsgesteuert. Im Vergleich zum allgemeinen Cursor-Konzept werden die verwendeten Datentypen an dieser Stelle eingeschränkt. Die in diesem Kapitel aufgeführten Operatoren arbeiten mit den Datentypen Zeichenkette (String), relationales Tupel und XML Dokument. Innerhalb von Tupeln oder XML Dokumenten sind allerdings weitere Datentypen wie Ganzzahlen erlaubt.

XML Dokumente werden zwischen den Anfrageoperatoren in Form von Hauptspeicher-DOM-Bäumen transferiert. Zu jedem Dokument ist die Speicherung von Metadaten möglich. Dies ist ein deutlicher Unterschied zu den relationalen Operatoren von XXL, wo Metadaten nur für Operatoren, nicht jedoch für einzelne Tupel spezifiziert werden können. Beispiele für Metadaten sind der Dokumentname, der Ursprung des Dokuments (Dateinamen, URL, etc.) oder anwendungsspezifische Zusatzinformationen.

Die in Kapitel 5 aufgeworfene Frage, ob die XML Verarbeitung für wenige große oder für viele kleine Dokumente optimiert sein sollte, wird bei dieser Anwendung eindeutig zugunsten des zweiten Falls beantwortet. Dokumente im Internet sind meistens vergleichsweise klein, damit die Ladezeiten beim Anwender erträglich bleiben. Somit ist die Speicherung von Dokumenten während der Anfrageverarbeitung prinzipiell auf einfache Art und Weise möglich.

Zentral für die Anfrageverarbeitung sind Operatoren, die XML Dokumente auf neue XML Dokumente abbilden. Einige Operatoren von XXL können zu diesem Zweck direkt verwendet werden, wenn sie speziell parametrisiert werden.

Dies trifft beispielsweise auf Abbildungs- oder Filteroperatoren zu. Für die speziellen Einsatzzwecke von XML werden in dem Framework einige vordefinierte Funktionen und Prädikate angeboten, die allgemein von Interesse sind.

Generell wird im Folgenden zwischen drei verschiedenen Arten von Operatoren: Eingabeoperatoren (Quellen), Verarbeitungsoperatoren und Ausgabeoperatoren (Senken). Da wir jedoch von drei verschiedenen Grunddatentypen ausgehen, gibt es Operatoren, die sowohl für einen Typ eine Quelle als auch für einen anderen Typ eine Senke sind. Im Folgenden wollen wir die Situation aus der Sicht des Datentyps „XML Dokument“ betrachten und für diesen auf einige Beispieloperatoren der drei verschiedenen Arten genauer eingehen.

7.1.1 XML Quellen

In XXL gibt es bereits eine Reihe von verschiedenen Datenquellen, die im Zusammenhang mit der XML Anfragebearbeitung verwendet werden können. Dies sind beispielsweise relationale Daten oder auch Indexstrukturen. Für XML Operatoren gibt es eine Reihe von weiteren interessanten Datenquellen, die direkt unterstützt werden.

Zum einen existiert eine Anbindung an den in Kapitel 5 vorgestellten nativen XML Speicher. Aus Dokumenten können Teilbäume ausgewählt und in Ströme umgewandelt werden. Als Metadatum kann hier die Position innerhalb des Dokuments in Form eines lokalisierenden XPath-Ausdrucks verwendet werden (über die Problematik hiervon siehe Kapitel 5.1.1.3).

Die zweite Art Eingabeoperator, der `DocumentPartitionCursor`, liest ein großes XML Dokument mit einem SAX-Parser ein und liefert bedarfsgesteuert disjunkte Teilbäume als kleine DOM-Dokumente zurück. Da hier Ereignisse sowohl vom SAX-Parser, als auch vom Aufrufer erzeugt werden, ist eine Entkopplung nötig. Hierzu wird eine ähnliche Technik wie in Kapitel 4.1.2 eingesetzt, die gleichzeitig für die Parsersteuerung sorgt.

Ein weiterer Eingabeoperator erzeugt einen Strom von XML Dokumenten, die aus Dateien generiert werden. Dieser Operator gliedert sich in zwei Teile. Der

erste Teil ist ein Cursor, der die Dateinamen der Dokumente als Zeichenketten liefert. Hierzu kann der speziell vordefinierte `FilenameCursor` verwendet werden, welcher alle Dateinamen von Dokumenten ermittelt, die sich in einem vorgegebenen Verzeichnis befinden. Ein solcher Cursor wird dem zweiten Teil des Eingabecursors, einem Mapper, übergeben. Die Abbildungsfunktion des Mappers nimmt die Dateinamen und aus den entsprechenden Dateien die DOM-Bäume der Dokumente.

Beim Einlesen von Dokumenten aus dem Internet stellt sich derzeit häufig das Problem, dass viele HTML-Dateien nicht XML konform (bzw. XHTML-konform) sind. Für die Konvertierung solcher Dateien nach XML gibt es Hilfsprogramme wie JTidy [JTi01].

Der `URLInputCursor` bekommt als Parameter einen Cursor übergeben, der die URLs der zu lesenden Seiten als Zeichenketten enthält. Der Operator liest auf Anforderung eine URL aus, transformiert den Inhalt mittels JTidy in echtes XML und gibt anschließend den DOM-Baum als Dokument zurück.

Zur Lokalisation von interessanten Datenquellen werden im Internet i. d. R. Suchmaschinen eingesetzt. Im Rahmen des hier vorgestellten Frameworks wurde ein Operator implementiert, der vorgegebene Begriffe an die Suchmaschine Google verschickt und die durch JTidy bereinigten Antwortdokumente nach und nach weiter gibt. Aus diesen werden durch die Anwendung von speziellen Filtern die Adressen der referenzierten Dokumente ermittelt (siehe Abbildung 7.1). Hierzu wird ein `XPathCursor` verwendet, der XML Dokumente nimmt und die Ergebnisse einer XPath-Anfrage (hier: „//A/@href“) in Form von Cursorn von Strings weiterleitet[3]. Im Beispiel werden die Strings durch Anwendung eines `Sequentializer`s aus den Cursorn herausgezogen. Durch einen Filter werden Eigenreferenzen von Google eliminiert. Die gewünschten URLs stehen nun in einem Cursor zur Verfügung, der anschließend dem `URLInputCursor` übergeben werden kann.

Bereits an dieser Stelle ist ein Vorteil dieser Architektur zu erkennen: Änderungen in dem Aufbau der Seiten von Google können sehr schnell durch kleine Änderungen in den Anfrageplänen kompensiert werden.

[3]Dieser Cursor ist eine XML Senke und eine Quelle für Strings.

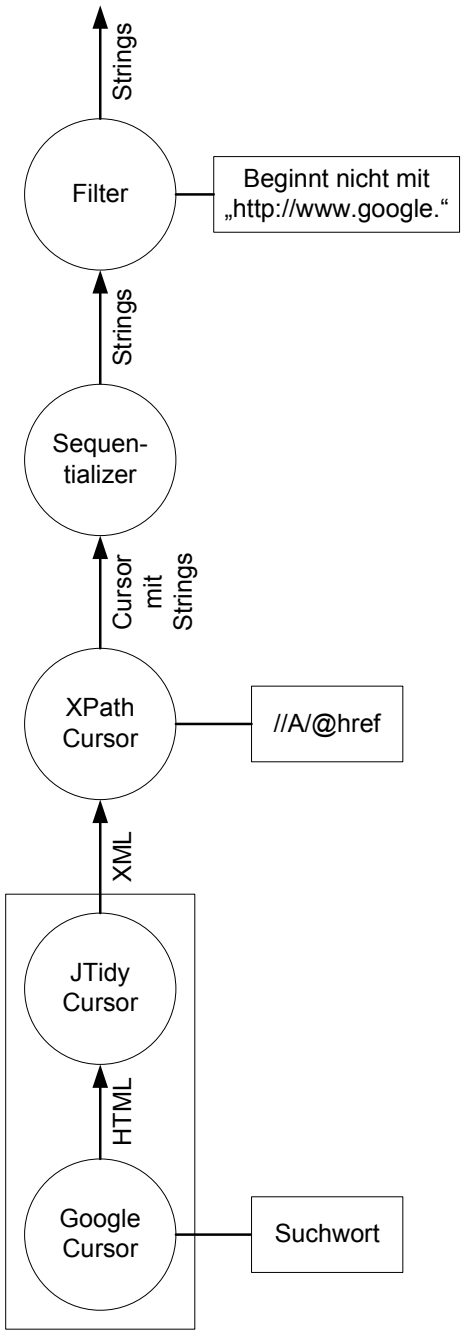


Abbildung 7.1: Anbindung von Google an XXL

7.1.2 XML Verarbeitungsoperatoren

Die im Folgenden vorgestellten Verarbeitungsoperatoren transformieren XML Dokumente in neue XML Dokumente. Auch die Metadaten der Dokumente können hierbei transformiert werden. Neben den bereits angesprochenen einfachen Operatoren wie Abbildungen oder Filter, gibt es einige Operationen, die speziell auf XML zugeschnitten sind und die Baumstruktur der Daten berücksichtigen.

XMLTK [ACGG+02] (XML Toolkit) ist eine Sammlung von Werkzeugen, die einige Verarbeitungsoperatoren für XML enthält. XMLTK bietet unter anderem Operatoren zur Gruppierung, zum Ver- und Entschachteln von XML Strukturen, sowie zur Paarbildung an. Die Operatoren werden jeweils durch eine Reihe von XPath-Ausdrücken parametrisiert. Ein Beispiel: Gruppiere in einem Dokument alle `book`-Elemente unterhalb von `/books` nach dem Erscheinungsjahr, welches in `/books/year/text()` zu finden ist. Verwende hierbei das Markup `booksInYear` als Elterntag der neuen Gruppen. Ein genauer Vergleich von XMLTK mit der entsprechenden Funktionalität von XXL folgt noch in Kapitel 7.4.

Die bisher angesprochenen Operatoren sind sehr grundlegend und wurden daher ebenfalls in XXL implementiert. Einige weitere Operatoren, die ausschließlich in XXL zu finden sind, sollen nun im Folgenden beschrieben werden.

Ein naheliegender, sehr mächtiger Operator, der `XSLTCursor`, führt eine Transformation mittels XSLT durch (siehe Abbildung 7.2). Als Parameter wird somit ein XSLT Programm übergeben. Mit diesem Operator können sehr viele verschiedenartige Transformationen durchgeführt werden, und zwar, ohne dass Programmiersprachenkenntnisse in Java benötigt werden.

In einigen Szenarien kann es sinnvoll sein, eine Validierung zeitlich versetzt nach dem Aufbau eines XML Dokuments durchzuführen. Hierfür gibt es den `ValidationCursor`. Er verwendet den Sun Multi-Schema XML Validator [Kaw03], der DOM-Hauptspeicherdokumente nachträglich validieren kann. Unterstützt werden verschiedene Arten von Grammatiken, unter anderem DTDs und XSchema Definitionen, aber auch das beliebter werdende Relax NG [Rel03].

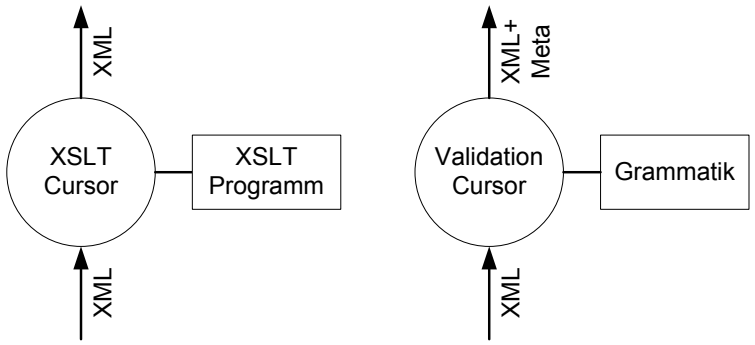


Abbildung 7.2: Beispiele für XML Verarbeitungsoperatoren in XXL

Der Validierungscursor führt selbst keine Filterung von nicht validen Dokumenten durch. Das liegt daran, dass es in einer Anwendung interessant sein kann, ausschließlich die Dokumente herauszufiltern, die eine vorgegebene Grammatik nicht erfüllen. Der Validierungscursor setzt deshalb nur das Metadatum namens „valid“, welches das Ergebnis der Validierung enthält, auf `true` oder `false`. Eine Applikation kann dann selbst entscheiden, welche Dokumente sie weiterverarbeiten will.

Die beiden zuletzt beschriebenen Cursor, der XSLT-Cursor und der Validierungscursor, sind beide Abbildungscursor (`Mapper`), die geeignet parametrisiert wurden. Die eigentliche Operation wird bei beiden durch die Abbildungsfunktion durchgeführt.

Ein weiterer Operator ist der `DiffCursor`. Dieser ist wichtig, wenn ähnliche Dokumente miteinander verglichen und Unterschiede festgestellt werden sollen. An Stellen, wo sich zwei Dokumente unterscheiden, werden vom `DiffCursor` spezielle Markuptags eingesetzt, unter die dann die unterschiedlichen Teilbäume der zwei Dokumente eingehängt werden.

Die große Stärke der Kombination des Operatoransatzes mit XXL ist jedoch, dass beliebige Verbundoperationen mit der vorhandenen XXL Operatoralgebra durchgeführt werden können. Die Verbundoperatoren werden hierbei durch spezielle Prädikate und Funktionen parametrisiert, von denen einige bereits vordefiniert sind. Ein Beispiel: Verbinde Dokumente, die Angaben über die gleiche

Stadt enthalten, und füge gewisse, spezifizierte Teilbäume der Dokumente in einem neuen Dokument unter einer vorgegebenen Wurzel zusammen.

Verbundoperationen können auch XML Ströme mit relationalen Daten in Beziehung setzen. Beispielsweise kann in einem Prädikat gefordert werden, dass der Pfadausdruck `/books/book/author/text()` mit dem Wert des Attributs `name` einer Relation `Personal` übereinstimmen muss. Das Ergebnis dieser Verbundoperation kann dann entweder ein XML Dokument sein, das an einer vorgegebenen Stelle das relationale Tupel in XML Darstellung enthält, oder ein Tupel, das um Informationen aus dem XML Dokument angereichert wurde.

7.1.3 XML Senken

Eine Applikation kann die Resultate, die XML Cursor liefern, entweder direkt verarbeiten oder aber vorimplementierte Verfahrensweisen nutzen. Die erste vordefinierte XML Senke ist der native XML Speicher von Kapitel 5 (siehe Abbildung 7.3). Die Resultate eines Operators können entweder als neue Dokumente oder als Teil von vorhandenen Dokumenten eingefügt werden. Im zweiten Fall muss zu jedem XML Dokument ein XPath-Ausdruck in den Metadaten vorhanden sein, der die Einfügeposition bestimmt.

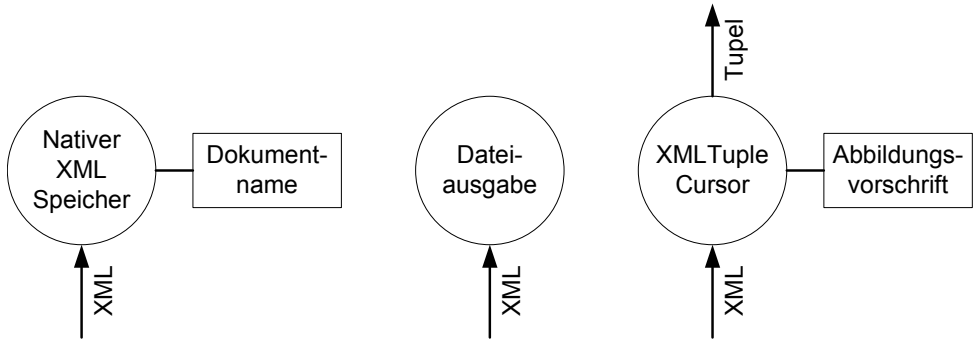


Abbildung 7.3: XML Senken in XXL

Ein Content Management System kann mit dieser Senke beispielsweise die folgende Verarbeitung durchführen. Es fordert eine Reihe von Dokumentteilen aus dem nativen Speicher an. Auf dem zurückgegebenen XML Strom führt es

die gewünschten Operationen aus und schreibt anschließend einige der Dokumentteile wieder in den nativen Speicher zurück.

Die zweite, implementierte Senke verfährt ganz ähnlich, nur dass sie die Dokumente in Dateien schreibt. Auch diese Senke kann ein Content Management System gut verwenden, beispielsweise, um statische XHTML-Seiten zu generieren, die anschließend in einem Webverzeichnis abgelegt werden.

Als dritte und letzte Senke soll hier noch der `XMLTupleCursor` erwähnt werden. Er wandelt XML Dokumente in Tupel um, indem er eine oder mehrere XPath-Anfragen ausführt und aus den Ergebnissen nach einer Abbildungsvorschrift ein Tupel generiert. Die Tupel können dann mit den Boardmitteln von XXL weiter verarbeitet werden. Eine mögliche Verarbeitung ist die Speicherung in einem relationalen Datenbanksystem unter Verwendung von JDBC.

7.2 Webservices

Eine weitere, wichtige Technik im Internet werden in Zukunft Webservices [WS004] sein. Sie dienen dazu, gezielt auf Informationen zuzugreifen, die meistens strukturiert, aber im Allgemeinen nicht relational sind. Ein Webservice wird im Internet über eine URL identifiziert. Beim Aufruf mittels des SOAP-Protokolls [SOA04][4] können dem Webservice Parameter übergeben werden, die er verarbeitet. Als Antwort liefert der Webservice ein XML Dokument zurück, das auf der Client-Seite mit Bibliotheksunterstützung ausgewertet wird.

Webservices sind selbstbeschreibend und können in öffentlich zugänglichen Webserviceverzeichnissen eingetragen werden (*Publish*). Durch solche Verzeichnisse ist es dem Anwender umgekehrt möglich, Webservices nach den dort bereitgestellten Metadaten zu suchen und zu finden (*Find*) und diese dann sofort in die eigene Verarbeitung einzubauen (*Bind* und *Use*).

Webservices sind so konzipiert, dass sie gut mit Anwendungsprogrammen kooperieren. Für eine direkte Benutzung durch Anwender sind sie dagegen nicht

[4] Um die Arbeit des Programmierers zu erleichtern, existieren einige Bibliotheken, die dafür sorgen, dass Webservice Aufrufe nicht nennenswert komplizierter sind als normale Methodenaufrufe.

ausgelegt. Die Benutzung von Webservices bietet Applikationen eine Reihe von Vorteilen. Durch sie können in standardisierter Weise aktuelle Informationen aus dem Web in die Verarbeitung integriert werden. Ein aufwendiges Parsen von Internetseiten mit Extraktion der gewünschten Information ist bei Ergebnissen von Webservices nicht mehr nötig. Falls ein Webservice ausfällt, kann dynamisch ein gleichartiger Service gesucht und im Erfolgsfall sofort benutzt werden. Weiterhin ist der Gedanke der dynamischen Erweiterbarkeit interessant: Eine Applikation kann ihre Funktionalität zur Laufzeit durch den Einsatz zusätzlicher Webservices erweitern.

Viele Firmen bieten schon jetzt Informationen über Webservices an. Zu diesen Firmen gehören beispielsweise Google und Amazon. Weitere typische Webservices liefern Wettervorhersagen (und zwar sowohl ortsgebundene, als auch überregionale Vorhersagen) oder die Echtzeitbörsenkurse. Momentan ist es häufig so, dass die frei verfügbaren Dienste hinsichtlich der Anzahl an Zugriffen begrenzt sind, die an einem Tag von einer IP-Adresse aus erfolgen dürfen (beispielsweise 1000 Suchergebnisse pro Tag und IP-Adresse). Bei einigen Anbietern ist es möglich, die uneingeschränkte Nutzung durch die Bezahlung einer monatlichen Gebühr zu erhalten. In Zukunft könnte dieses Modell Schule machen und im Internet zu einem verbreiteten Geschäftsmodell avancieren.

Die Anbindung von Webservices an XXL wurde bereits in einer Diplomarbeit [Mic03] unter meiner Anleitung durchgeführt. Diese Anbindung ist in Abbildung 7.4 dargestellt. Der Webservice Cursor erhält von einem Eingabecursor Daten (XML Dokumente oder aber Tupel mit Parameterbelegungen) und generiert hieraus eine SOAP-Anfrage. Diese sendet er an einen angegebenen Webservice. Dessen Antwort wird anschließend entgegengenommen und als XML Dokument einem Aufrufer weitergereicht.

Im Allgemeinen ist der Webservice Cursor also ein Verarbeitungsoperator. Wenn ein Webservice keine Eingabeparameter benötigt, so kann er mit einem speziellen Cursor parametrisiert werden, der eine vordefinierte oder beliebige Anzahl an Nullwerten liefert. In diesem Fall wird der Webservice Cursor als Datenquelle verwendet. Da es auch Webservices geben kann, die eine Speicherung oder eine selbstständige Weiterverarbeitung anbieten (z. B. bei einem Internet-

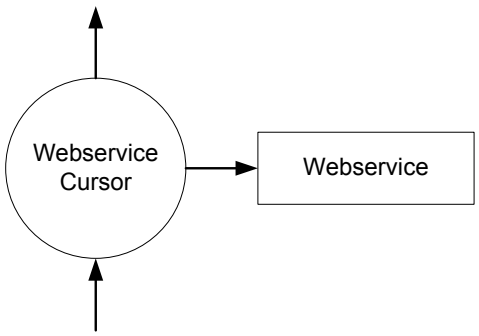


Abbildung 7.4: Anbindung von Webservices an die Cursor Algebra von XXL

Fotodienst), kann diese Art Cursor bei einer anderen Parametrisierung auch als reine Senke verwendet werden.

7.3 Beispielszenarien

Die beschriebenen Operatoren können beispielsweise dazu verwendet werden, große Teile von XQuery zu implementieren. Eine Beispielanfrage ist in Abbildung 7.5 zu finden.

```
for $b in document("www.bn.com/bib.xml")//book {
    $e in $b/*[contains(string(.),"Suciu")]
where ends_with(local-name($e),"or")
return
    <book>
        { $b/title }
        { $e }
    </book>
```

Abbildung 7.5: Eine Beispielanfrage in XQuery

In dieser Anfrage wird zunächst nach Dokumenten mit `book`-Element gesucht. Dies kann in XXL durch einen Eingabeoperator der nativen XML Speicherungsstruktur umgesetzt werden. In den gefundenen Dokumenten muss unterhalb einer `book`-Stelle ein weiteres Markuptag $e vorhanden sein, unterhalb dessen wiederum in beliebiger Tiefe die Zeichenkette „Suciu“ existieren muss.

Für die Umsetzung dieses Teils in XXL-Operatoren kann ein Filter mit XPath-Prädikat verwendet werden. Die XQuery-Anfrage fordert zusätzlich, dass das Tag $e auf „or“ enden muss (eine weitere Filteroperation in XXL). Das Resultat wird anschließend aus den gefundenen Dokumentstellen zusammengesetzt. Hierzu kann der Schachtelungsoperator verwendet werden.

Die Zuordnung von Variablen zu konkreten DOM-Knoten kann durch einen Eintrag in den Metadaten des entsprechenden Dokuments erfolgen. Somit sind für diese Anfrage außer den beschriebenen, parametrisierten Operatoren keine Zusatzstrukturen nötig.

Noch interessanter ist die Verwendung der beschriebenen Anfrageoperatoren jedoch für Daten aus dem Internet. Im Internet werden Daten in verschiedenen Formen angeboten. Es existiert relationales XML in Form von HTML-Tabellen, das in relationale Tupel umgewandelt werden kann (siehe Kapitel 4). Weiterhin gibt es im Internet echt semi-strukturierte Daten in Form von HTML- oder XML-Dokumenten und es gibt Webservices. Mittels der XXL Operatoren können Anfragen gestellt werden, die auf allen diesen Arten und zusätzlich auf lokal vorhandenen Daten aufsetzen.

Abbildung 7.6 zeigt ein Szenario, in dem Angebote für bestimmte Produkte bei ebay und Amazon über Webservices gesucht werden. Diese Angebote werden dann mittels weiterer Operatoren miteinander verglichen. Interessant wären beispielsweise diejenigen Artikel, die bei ebay momentan für weniger als 50% des Amazon Preises angeboten werden.

Weiterhin könnte hier noch eine Zusatzbedingung eingebaut werden, nämlich, dass Artikel von ebay nur dann betrachtet werden sollen, wenn deren Auktion in den nächsten x Minuten ausläuft.

Bislang sind die vorgestellten, neuen Operatoren nur aus Java heraus einsetzbar. Ein Anwender würde allerdings gerne eine einfachere Nutzungsmöglichkeit erhalten. Die Definition einer deklarativen Sprache ist für diesen Zweck nicht sinnvoll, da eine solche Sprache zu unflexibel für die regelmäßig neu auftauchenden Anforderungen wäre. Daher wurde für XXL ein allgemeineres Verfahren erdacht. Eine GUI (Graphical User Interface) [KS04] erleichtert die Erstellung

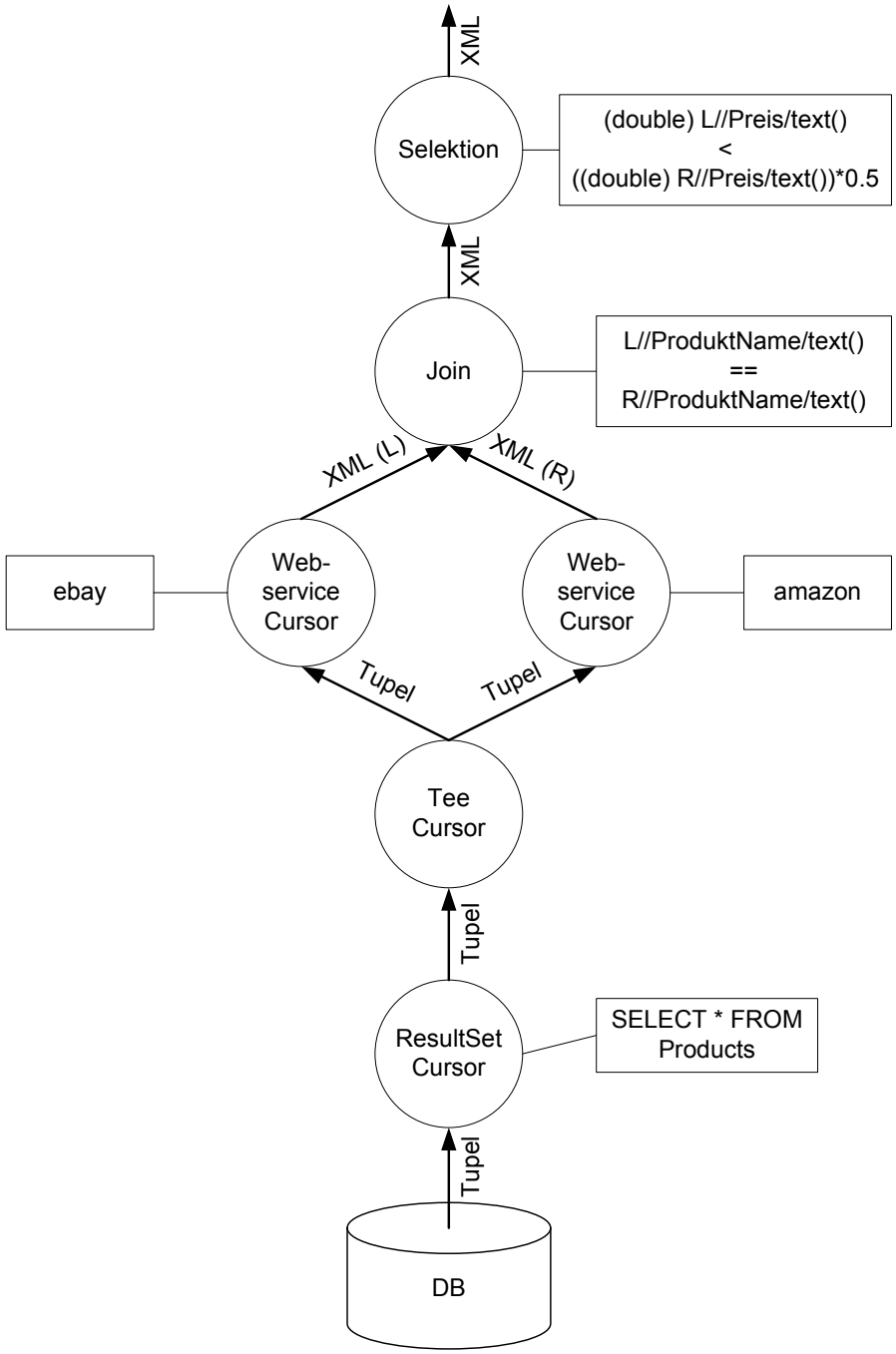


Abbildung 7.6: Ein Szenario für die Verarbeitungsoperatoren mit Verwendung von Webservices

von beliebigen Operatorbäumen. Dank des Java Reflection Mechanismus können vorgegebene Klassen visuell über ihre Konstruktoren miteinander verdrahtet werden. Somit ist beispielsweise die visuelle Erzeugung von Anfragegraphen möglich. Zusätzlich erzeugt die GUI automatisch den Quelltext für diese Graphen, übersetzt diesen und führt ihn auf Wunsch des Anwenders aus.

Damit die GUI sinnvoll einsetzbar ist, werden ausreichend vorgefertigte Funktionen und Prädikate benötigt. Hieran wird momentan noch gearbeitet, damit dem künftigen Anwender eines solchen Systems eine ausreichende Funktionalität zur Verfügung gestellt werden kann.

7.4 Literatur

XML Operatoren wurden, wie bereits angesprochen, im Rahmen der XMLTK-Bibliothek vorgeschlagen. Diese Bibliothek wurde in Washington D.C. unter der Leitung von Dan Suciu [ACGG+02] entwickelt. XMLTK enthält Operatoren, die von der Kommandozeile aus aufgerufen werden und XML Dateien in neue XML Dateien umwandeln. Die Idee hierzu stammt von den Kommandozeilenwerkzeugen von Unix (kurz: Unixtools), welche eine effektive Manipulation von Textdateien erlauben. Die Unixtools werden in der Praxis für viele Zwecke eingesetzt, beispielsweise für die Bearbeitung von Konfigurationsdateien. Da inzwischen immer mehr Konfigurationsdateien einen XML Syntax besitzen, liegt der Gedanke nahe, entsprechende Werkzeuge für XML zu entwickeln.

In XMLTK werden eine Reihe verschiedener Operatoren vorgeschlagen: zum Sortieren von XML, zur Aggregation, zur Ver- und Entschachtelung (Nest und Flatten), zur Paarbildung und zur Ermittlung des Anfangs bzw. des Endes eines Dokuments (Head und Tail). Um die Auswertung der XML Dateien zu beschleunigen, setzt XMLTK eine binäre Repräsentation von XML ein. Diese wird bei jedem Eingabeoperator erzeugt. Bei den Ausgabeoperatoren wird die binäre Repräsentation wieder in gewöhnliches XML transformiert. Beim Aufbau einer binären Repräsentation wird außerdem ein Index namens SIX (Stream IndeX) angelegt. Er erlaubt es, nicht benutzte Teile von Dokumen-

ten gezielt zu überspringen und somit einen Teil der E/A-Zugriffe einzusparen. Hierdurch wird ein erneutes, aufwendiges Parsen vermieden.

XMLTK stellt seine Operatoren leider nur per Kommandozeile zur Verfügung. Aus Sicht eines Anfragesystems ist dies prinzipiell unbefriedigend, da die Anfragen nicht gesteuert werden können. Die Verarbeitung der Dokumente erfolgt grundsätzlich Operator für Operator und nicht Dokument für Dokument. Zuerst arbeiten die Operatoren direkt über den Quellen, danach die Operatoren eine Ebene höher usw. bis das Endergebnis komplett generiert wurde. Eine bedarfsgesteuerte Verarbeitung liegt also nicht vor. Hierdurch ist dieser Ansatz nicht direkt für die Anfrageverarbeitung im Internet einsetzbar. Außerdem sind die Kosten durch die Zwischenspeicherung teilweise enorm hoch.

Aus den genannten Gründen wird in XXL mit einer Hauptspeicherrepräsentation der Dokumente gearbeitet. Sie ermöglicht einen bedarfsgesteuerten Austausch zwischen den Operatoren. Bei dem betrachteten Szenario mit vielen kleinen Dokumenten ist der benötigte Hauptspeicherbedarf relativ gering. Somit ist die vorgeschlagene Verarbeitung problemlos möglich.

Eine weitere Arbeit soll an dieser Stelle noch Erwähnung finden. ActiveXML [ABM+02] ist eine Technologie, die bei INRIA in Paris erdacht wurde. Das Konzept integriert aktive Anteile in die ansonsten passiven XML Dokumente. Aktive Anteile sind Beschreibungen von Webserviceaufrufen, die bei Bedarf ausgelöst werden können (vom Anwender oder automatisch). Die Resultate der Aufrufe, die wiederum aktive und passive Inhalte besitzen können, werden anschließend an eine vordefinierte Stelle in das Dokument zurückgeschrieben.

Auf dieser Basis wurde bei INRIA ein Peer-to-Peer Auktionsszenario implementiert. Generell eignet sich ActiveXML gut für Applikationen, bei denen Daten zusammengetragen werden sollen. Bis zu sinnvollen AXML-Anwendungen ist jedoch eine weitere Verbreitung des Datenformats im Internet eine unbedingte Voraussetzung.

Der ActiveXML Ansatz ist für XXL interessant, weil er mit den zur Verfügung gestellten Operatoren dieser Arbeit implementiert werden kann. Somit zeigt sich ein weiteres Mal die Nützlichkeit dieses allgemeinen und bislang einzigartigen Operatoransatzes.

7.5 Zusammenfassung und Ausblick

In diesem Kapitel wurden Operatoren vorgeschlagen, die auf den Datentypen String, Tupel und XML Dokument arbeiten. Kern der Anfrageverarbeitung sind XML Operatoren, die XML Dokumente auf neue XML Dokumente abbilden. Diese können in der Praxis bei vielen interessanten Anfragen in Applikationen verwendet werden, sei es zur Implementierung von XQuery oder zur Beantwortung von Anfragen auf Internetdaten. Die Anbindung an Webservices stellt eine weitere wichtige Grundlage für die Anfrageverarbeitung dar.

Die entwickelten XML-Operatoren setzen auf dem Cursor-Konzept von XXL auf und sind dementsprechend bedarfsgesteuert implementiert. Durch die Verwendung von DOM als Repräsentation für Dokumente können in den Operatorbäumen genau wie bei relationalen Systemen Pipeliningeffekte ausgenutzt werden.

In Zukunft sollen in XXL noch einige weitere Operatoren hinzugefügt werden. Außerdem soll durch weitere vordefinierte Funktionen und Prädikate die Anbindung an die XXL GUI verbessert werden.

Das Themengebiet der Anfrageverarbeitung auf Internetdaten ist generell sehr vielschichtig. In diesem Bereich muss sicherlich noch viel Forschungsarbeit geleistet werden, insbesondere, was die Gewinnung von strukturierten Daten aus einfachen Texten angeht. Die in diesem Kapitel beschriebenen Konzepte eröffnen schon heute einen Weg, mit dem interessante, dokumentübergreifende Anfragen für Jedermann möglich werden. Sowohl für Applikationsprogrammierer, als auch für Anwender kann das erstellte Framework in Zukunft von hohem Nutzen sein.

Kapitel 8

Fazit und Ausblick

In der Mythologie wird Bäumen seit je her ein gutes Erinnerungsvermögen zugesprochen. Der Volksmund behauptet, Bäume können sich an unsere Vorfahren erinnern [Röh88]. Bäumen werden auch noch weitere positive Eigenschaften zugeschrieben, wie z. B. dass sie die Zeiten überdauern und sich bei Bedarf regenerieren. Selbst in Werken der zeitgenössischen Musik spiegeln sich diese Themen wieder, beispielsweise in „The Memory of Trees“ von Enya.

Auch in Datenbanksystemen wird seit je her auf Baumstrukturen zurückgegriffen. So wie das Waldsterben und die Umweltschutzbewegung zu einem neuen Baumbewusstsein in der Gesellschaft führten, so ist es im Datenbankbereich möglich, dass das Auftreten von XML zu einem Umdenken führt. Dies könnte eine Erweiterung des häufig auf strukturierte Daten beschränkten Horizonts bewirken.

Von einigen Personen wird bereits das Zeitalter der XML Datenbanksysteme beschworen. Selbst, wenn diese Aussage zu weit führt, so wird die Notwendigkeit zur Verwaltung von XML in Datenbanksystemen [McG01] allgemein anerkannt. Momentan fehlt den meisten angebotenen XML verwaltenden Systemen jedoch noch einiges von der tatsächlich benötigten Funktionalität. Bei nativen XML Datenbanksystemen werden momentan noch wichtige Sprachen für die Datenmanipulation und für datenbankspezifische Aufgaben vermisst, wie für die Anwenderverwaltung und für die Indexerstellung. Dies ist für die Verbreitung von XML Datenbanksystemen in der Wirtschaft sehr nachträglich. Weiterhin stellt sich die Frage, wie native XML Datenbanksysteme von

denjenigen angenommen werden, die damit täglich Umgang haben: von den Applikationsprogrammierern.

In diesem abschließenden Kapitel soll zunächst ein Fazit dieser Arbeit gezogen werden. Danach wird auf einige interessante Punkte für künftige Forschungsarbeiten im Bereich von XML Datenbanksystemen eingegangen. Hierzu zählt nach meiner Einschätzung nach ganz grundlegend die Implementierung einer effizienten Ähnlichkeitssuche auf nativem XML Speicher.

8.1 Fazit

Im Rahmen dieser Arbeit wurde ein Teil eines nativen XML Datenbanksystems implementiert und wesentliche Komponenten davon evaluiert. Dabei wurde besonderer Wert auf eine solide Fundierung gelegt. Die Evaluierung erfolgte sowohl anhand eines Kostenmodells als auch durch praxisrelevante Laufzeitmessungen.

Die komplett neu erstellte Infrastruktur ist in XXL für alle interessierten Personen verfügbar (LGPL-Lizenz). In sie wurden im Rahmen dieser Arbeit zum einen viele Verfahren aus der Literatur und zum anderen eine Reihe von neuen Verfahren implementiert. Hierzu zählen beispielsweise neue Strategien für den Recordmanager. Im Bereich der Abbildung von relationalem XML auf relationale Datenbanken wurden Verfahren zum bedarfsgesteuerten Lesen entwickelt.

Das Hauptthema dieser Arbeit waren native Speicherungsstrukturen für XML Datenbanksysteme. Hierzu wurde ein Framework implementiert, das in seinen Grundstrukturen dem XML Datenbanksystem Natix ähnelt, jedoch sehr viel flexibler ist. Ein spezielles Augenmerk wurde auf Verfahren zur Verwaltung von dynamisch wachsenden oder schrumpfenden XML Dokumenten gelegt. Die Evaluierung der in diesem Framework implementierten Onecut Splitalgorithmen zeigte, dass diese Verfahren die bisher bekannten Verfahren um bis zu 65 Prozent in der Leistung übertreffen.

Für die Verbesserung der Anfrageleistung wurde in die Speicherungsstruktur ein Signaturindex integriert. Dies hat sich bei Bulkloading als sehr vorteil-

haft für die Beantwortung von XPath-Anfragen erwiesen. Hierbei können fast 30% der Externspeicherzugriffe eingespart werden, wohingegen der Baumaufbau praktisch ohne Mehraufwand auskommt.

Bei der relationalen Speicherung von XML war bislang nicht bekannt, wie sich die verschiedenen Ansätze aus der Literatur im Vergleich zu nativen XML Strukturen behaupten. Erstmals konnten im Rahmen dieser Arbeit entsprechende Laufzeitvergleiche innerhalb eines einzigen Systems und unter Verwendung identischer Basiskomponenten durchgeführt werden. Sie lassen erkennen, dass die Abbildung von XML Dokumenten auf Relationen häufig Nebeneffekte wie unflexible Verarbeitung oder einen hohen Speicherplatzbedarf besitzen. Hinsichtlich aller Anforderungen in einem nativen XML Datenbanksystem zeigte sich Bulkloading als bestes Verfahren im Test.

Interessant ist der Ansatz der XML Ströme zur Anfrageverarbeitung. Hier geht man davon aus, dass viele kleine XML Dokumente verwaltet werden müssen. Dies ist die Situation, wie sie beispielsweise im World Wide Web anzutreffen ist. In der Arbeit wurden einige neue Operatoren vorgeschlagen, die für die Beantwortung von Anfragen gut eingesetzt werden können. Alle Operatoren wurden als bedarfsgesteuerte Cursor in XXL implementiert und in einem flexiblen Anfrageframework zur Verfügung gestellt.

Hauptergebnisse dieser Arbeit sind zum einen die angesprochenen Verbesserungen für die Basisstrukturen von nativen XML Datenbanksystemen. Zum anderen stellt die frei verfügbare Implementierung von nativem XML Speicher eine Bereicherung für die Forschergemeinde dar. Die Implementierung ist die erste frei verfügbare Implementierung der Natix-Ideen, die ich für sehr grundlegend und wichtig halte. Anderen Forschern kann XXL nun dazu dienen, eigene Verfahren mit diesen Ideen zu vergleichen. Hierzu ist nun nicht mehr die Implementierung eines komplett neuen Systems nötig.

8.2 Die Zukunft von XML Datenbanksystemen

Wie sieht die weitere Zukunft von XML Datenbanksystemen aus? Das Schicksal der letzten großen Generation von neuartigen Datenbanksystemen, den ob-

jektorientierten Datenbanksystemen, regt hier zum Nachdenken an. Rein objektorientierte Datenbanksysteme sind heutzutage nahezu ausgestorben. Wenn überhaupt, dann werden sie in Forschung und Lehre eingesetzt.

Die großen Datenbanksystemhersteller Oracle, IBM und Microsoft haben sich viele Ideen von diesen Systemen abgeschaut und in ihre eigenen Systeme integriert. Hieraus entstanden die objektrelationalen Datenbanksysteme. Heutzutage wird an vielen Stellen behauptet, dass die Technik von reinen objektorientierten Datenbanksysteme im Grunde genommen besser und sauber ist als bei den marktbeherrschenden Systemen. Die Semantik der Standards, auf denen sie basieren, ist klarer, deren Definition kürzer und außerdem sind reale Systeme teilweise sogar performanter. Sie sind weiterhin besser an die Problemstellungen der Praxis angepasst. Die Entwicklung einer Datenbankapplikation kann hier vollständig objektorientiert erfolgen, und eine aufwendige Abbildung von Objekten in das relationale Modell ist überflüssig.

Doch die Marktmacht der großen Hersteller ließ damals keine Technologie zu, bei der sie nicht selbst involviert waren. Ähnlich sieht es aktuell bei XML aus. Immer mehr Fähigkeiten werden in die großen Datenbanksysteme integriert. Die großen Firmen entwickeln ihre XML befähigten Systeme weiter, so dass objektrelationale Datenbanksysteme mit integriertem XML Datenbanksystem entstehen. Allein, wie diese Systeme benannt werden sollen, ist noch ungeklärt.

Prinzipiell sind integrative Systeme auch sinnvoll. Das Hauptanliegen eines Applikationsprogrammierers ist, dass er bestimmte Daten in einer Datenbank verwalten möchte. Diese Daten können sowohl unstrukturiert, semi-strukturiert als auch voll strukturiert sein. Wichtig ist, dass für alle diese Daten möglichst nur ein einziges System benötigt wird, da ansonsten die Möglichkeiten der kombinierten Anfrageverarbeitung nicht gegeben sind und in der Folge die Komplexität auf Applikationsseite steigt.

Die dafür benötigten OR-XML Datenbanksysteme sind jedoch riesig und komplex, weshalb sie nicht von heute auf morgen entstehen. Nur große Firmen können solche Systeme überhaupt anbieten. Durch die Notwendigkeiten der Praxis könnte in Zukunft die Marktmacht der großen Anbieter im professionellen Umfeld noch weiter gefestigt werden.

Eine weitere Gefahr für XML-verarbeitende Systeme besteht in den teilweise noch nicht vorhandenen oder nicht akzeptierten Standards zur Datendefinition und Datenmanipulation. Wenn dies so bleibt, ist es fraglich, ob Applikationen auch in Zukunft immer noch einigermaßen unabhängig vom verwendeten System entwickelt werden können.

Speziallösungen für XML werden sich nur dort durchsetzen können, wo wenig Geld für große Softwaresysteme vorhanden ist oder wo wirklich nur eine ganz spezielle Funktionalität benötigt wird. Viele Webapplikationen werden sicherlich mit einem reinen XML Datenbanksystem auskommen können. Hier gibt es also einen Markt, der einen Überlebenskorridor für reine XML Datenbanksysteme schaffen könnte.

8.3 Ähnlichkeitssuche

Was Anfragesprachen für XML angeht, so liegt das Augenmerk der interessierten Fachwelt seit längerem auf der Definition von XQuery, die demnächst endgültig erfolgen wird. Allerdings bietet XQuery nicht alle Möglichkeiten, die von den Anwendern gewünscht werden. Das zeigte bereits Kapitel 7 zu XML Strömen.

Zu dem Zeitpunkt, an dem ein Anwender eine Anfrage stellt, ist bei echt semistrukturierten Daten kein exaktes Schema der Daten vorhanden. Dies wird allerdings für sinnvolle Anfragen in XQuery vorausgesetzt. Somit eignet sich XQuery zwar hervorragend für vollstrukturierte XML Dokumente mit Schema, jedoch nur sehr bedingt für echt semi-strukturierte Daten.

Einen bislang zu kleinen Stellenwert besitzen in der aktuellen Diskussion Anfragesprachen für Ähnlichkeitssuche [STW01]. Der Anfragende weiß hierbei nicht genau, wie die Daten aussehen. Er weiß aber, dass in den Daten etwas stehen könnte, das ihn interessiert. Der Suchende stellt seine Anfrage nun in Form eines kleinen beispielhaften XML Dokumentes. Dieses wird mit den Dokumenten der Datenbank auf Ähnlichkeit hin überprüft.

Von entscheidender Bedeutung ist hier das Ähnlichkeitsmaß. Es können verschiedene Techniken des *tree matchings* [ZS97] eingesetzt werden. Der Nachteil ist hierbei der hohe Rechenaufwand. Je nach Art des *tree matchings* kann die Rechenzeit quadratisch mit der Dokumentgröße ansteigen. Dies gilt jedoch nur für den Fall, dass die zu vergleichenden Dokumente in etwa gleiche Größe besitzen. Ansonsten ist der Aufwand proportional zum Produkt der Größe der Eingabedokumente. Da die Anfrage im Vergleich zu den Dokumenten der Datenbank sehr klein ist, bleibt eine Hoffnung auf tolerierbare Antwortzeiten.

Um die Antwortzeiten weiter verkürzen zu können, ist es sinnvoll, Abschätzungen für die Ähnlichkeitsfunktion vorzunehmen. Häufig können untere und obere Grenzen effizient berechnet werden, was zum Abschneiden von großen Teilbäumen aus dem Suchraum führen kann [GJK+02].

Durch den in dieser Arbeit implementierten Signaturindex für nativen XML Speicher, können solche Abschätzungen sehr viel genauer erfolgen, als dies in anderen Systemen bislang effizient möglich war. Es bietet sich an, diesen Ansatz weiter zu verfolgen.

8.4 Ausblick

Das im Rahmen dieser Arbeit implementierte Gerüst bietet bereits eine ganze Menge an Funktionalität, die für ein natives XML Datenbanksystem benötigt wird. Jedoch wäre für ein vollständiges System noch viel Aufwand nötig. Hierzu müssten noch einige Dienste und mindestens XQuery als Anfragesprache implementiert werden.

Aufgrund verschiedener Faktoren ist nicht anzunehmen, dass eine Weiterentwicklung in Richtung auf ein kommerzielles System erfolgversprechend, sprich finanziell lukrativ wäre. Auf dem Markt für XML Datenbanksysteme gibt es bereits sehr viel Konkurrenz, und künftig werden wie angesprochen die großen Datenbanksystemhersteller noch mehr in diesen Markt hineindrängen. Diese Firmen haben ein sehr viel größeres Potential an Programmierern als dies an einer Universität der Fall ist.

Ein kommerzielles Produkt sollte weiterhin auf einer hardwarenäheren Programmiersprache als Java basieren, weil hierdurch das System leistungsfähiger sein kann.

Dennoch ist das XML Gerüst ein wichtiger Bestandteil der XXL Bibliothek. Forscher erhalten hierdurch die Möglichkeit, mit nativem XML Speicher Erfahrungen zu sammeln, neue Ideen einzubringen und zu testen. Es ist also der Grundstock vorhanden, der für den Test von höherwertigen Funktionen eines XML Datenbanksystems unbedingt benötigt wird.

Anhang A

Anmerkungen zum Quelltext

Der komplette Quelltext, der dieser Arbeit zugrunde liegt, wird, soweit dies nicht bereits geschehen ist, Teil der nächsten Version der *eXtensible and fleXible library (XXL)* sein.

Die Dokumentation der erstellten Klassen ist umfangreich und in englischer Sprache verfasst. Die Mühen dieser Dokumentation wären für die Erstellung dieser Arbeit sicherlich nicht unbedingt nötig gewesen. Dieser Weg wurde trotzdem gewählt, um Wissenschaftlern einen einfachen Zugang zu diesen Methoden zu ermöglichen.

Forschung muss auch dafür sorgen, dass Tests nachvollziehbar und im Detail kritikfähig sind. Dies wird durch die Offenlegung und Nutzbarmachung des Quelltextes erreicht.

Ich hoffe, dass viele Wissenschaftler diese Bibliothek als Grundlage für die Implementierung eigener Verfahren verwenden. XXL bietet für viele Zwecke eine gute und breite Basis an Komponenten. Hierdurch sinkt die Entwicklungszeit für neue Verfahren, und Ideen können schneller auf ihre Tauglichkeit hin überprüft werden.

Literaturverzeichnis

[ABM+02] Serge Abiteboul, Omar Benjelloun, Ioana Manolescu, Tova Milo und Roger Weber: *Active XML: Peer-to-Peer Data and Web Services Integration.* In: *Proc. of the VLDB*, Seiten 1087–1090, 2002.

[ABS99] Serge Abiteboul, Peter Buneman und Dan Suciu: *Data on the Web: From Relations to Semistructured Data and XML.* Data Management Systems. Morgan Kaufmann, 1999.

[ACFR02] Serge Abiteboul, Sophie Cluet, Guy Ferran und Marie-Christine Rousset: *The Xyleme Project.* Comput. Netw., 39(3):225–238, 2002.

[ACGG+02] Iliana Avila-Campillo, Todd J. Green, Ashish Gupta, Makoto Onizuka, Demian Raven und Dan Suciu: *XMLTK: An XML Toolkit for Scalable XML Stream Processing.* In: *Programming Languages Technologies for XML (PLANX)*, Oktober 2002.

[ACV+02] Vincent Aguilera, Sophie Cluet, Pierangelo Viltri, Dan Vodislav und Fanny Wattez: *Querying XML Documents in Xyleme.* Seiten 79–94, 2002.

[Agu04] Vincent Aguilera: *XOQL Homepage.* Online im Internet [Stand: September 2004], 2004. http://www-rocq.inria.fr/∽aguilera/xoql.

[ANS86] ANSI: *Database Language SQL, Document ANSI X3.135*, 1986. Also available as: International Standards Organization Document ISO/TC97/SC21/WG 3 N 117.

[AS02] Sihem Amer-Yahia und Divesh Srivastava: *A mapping schema and interface for XML stores.* In: *Fourth ACM CIKM International Workshop on Web Information and Data Management (WIDM 2002)*, Seiten 23–30. ACM, 2002.

[BA03] Robert Blumberg und Shaku Atre: *The Problem with Unstructured Data*, 2003. http://www.DMReview.com.

[BBD+01] Jochen van den Bercken, Björn Blohsfeld, Jens-Peter Dittrich, Jürgen Krämer, Tobias Schäfer, Martin Schneider und Bernhard Seeger: *XXL - A Library Approach to Supporting Efficient Implementations of Advanced Database Queries.* In: *Proc. of the VLDB*, Seiten 39–48, 2001.

[BDB04] *Berkeley DB.* Online im Internet [Stand: September 2004], 2004. http://www.sleepycat.com.

[BDS00] Jochen van den Bercken, Jens-Peter Dittrich und Bernhard Seeger: *javax.XXL: A prototype for a Library of Query processing Algorithms.* In: *Proc. of the ACM SIGMOD*, Seite 588, 2000.

[Bos99] John Bosak: *The Plays of Shakespeare*, 1999. http://www.ibiblio.org/bosak.

[Bou04] Ronald Bourret: *XML programming, writing and research.* Online im Internet [Stand: September 2004], 2004. http://www.rpbourret.com.

[Bra04] Tom Bradford: *Getting Reacquainted with dbXML 2.0.* Online im Internet [Stand: September 2004], 2004. http://www.xml.com/pub/a/2004/02/25/dbxml.html.

[BSM+03] Chris Brandin, Harald Schöning, Wolfgang M. Meier, John Merrells, Michael Olson, Shawn E. Benham, Uwe Hohenstein, Michael Rys, Richard Edwards, Paul G.

Brown, Jr. Harry G. Direen, Mark S. Jones, Lee Anne Kowalski, Karen Eglin, Lily Hendra, Odysseas Pentakalos, Ruth Wilson, Maria Cobb, Frank McCreedy, Roy Ladner, David Olivier, Todd Lovitt, Kevin Shaw, Fred Petry, Mahdi Abdelguerfi, David C. Rine, Rosa Meo, Giuseppe Psaila, Xavier Baril, Zohra Bellahsène, Stéphane Bressan, Mong Li Lee, Ying Guang Li, Zoé Lacroix, Ullas B. Nambiar, Jignesh M. Patel, H. V. Jagadish, Cosima Schmauch, Torsten Fellhauer, Joseph Fong, H. K. Wong und Anthony Fong: *XML Data Management. Native XML and XML-enabled Database Systems.* Addison-Wesley Professional, 2003.

[Bun97] Peter Buneman: *Semistructured Data.* In: *Proceedings of the Sixteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Seiten 117–121. ACM Press, Mai 1997.

[Cas04] *Castor*, 2004. http://www.castor.org.

[CLO03] Qun Chen, Andrew Lim und Kian Win Ong: *D(k)-index: an adaptive structural summary for graph-structured data.* In: *Proc. of the ACM SIGMOD*, Seiten 134–144. ACM Press, 2003.

[CN04] James Cheng und Wilfred Ng: *XQzip: Querying Compressed XML Using Structural Indexing.* In: Elisa Bertino, Stavros Christodoulakis, Dimitris Plexousakis, Vassilis Christophides, Manolis Koubarakis, Klemens Böhm und Elena Ferrari (Herausgeber): *Proc. Int. Conf. on Extending Data Base Technology*, Band 2992 der Reihe *Lecture Notes in Computer Science*. Springer, 2004.

[Cod70] E. F. Codd: *A Relational Model of Data for Large Shared Data Banks.* Communications of the ACM, 13(6):377–387, 1970.

[Com79] Douglas Comer: *Ubiquitous B-Tree.* ACM Computing Surveys, 11(2):121–137, 1979.

[Coo02] Dave Cook: *Inside The Windows 2000 Registry.* Online im Internet [Stand: September 2004], März 2002. http://www.pcsupportadvisor.com.

[CS03] Surajit Chaudhuri und Kyuseok Shim: *Storage and Retrieval of XML Data using Relational Databases.* Seite 802. IEEE Computer Society, 2003.

[Dep86] Uwe Deppisch: *S-Tree: A Dynamic Balanced Signature Index for Office Retrieval.* In: *SIGIR'86, Proceedings of the 9th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, Pisa, Italy, September 8-10, 1986*, Seiten 77–87. ACM, 1986.

[DFS99] Alin Deutsch, Mary F. Fernandez und Dan Suciu: *Storing Semistructured Data with STORED.* In: *Proc. of the ACM SIGMOD*, Seiten 431–442. ACM Press, 1999.

[DOM] *Document Object Model (DOM).* Online im Internet [Stand: September 2004]. http://www.w3.org/DOM.

[EBC+03] E. Eleftheriou, P. Bächtold, G. Cherubini, A. Dholakia, C. Hagleitner, T. Loeliger, A. Pantazi, H. Pozidis, T. R. Albrecht, Gerd K. Binnig, Michel Despont, Ute Drechsler, Urs Dürig, B. Gotsmann, D. Jubin, Walter Häberle, M. A. Lantz, Hugo E. Rothuizen, Richard Stutz, Peter Vettiger und D. Wiesmann: *A Nanotechnology-based Approach to Data Storage.* VLDB, Seiten 3–7, 2003.

[eXi04] *Open Source Native XML Database.* Online im Internet [Stand: September 2004], 2004. http://exist.sourceforge.net.

[FHK+02] Thorsten Fiebig, Sven Helmer, Carl-Christian Kanne, Guido Moerkotte, Julia Neumann, Robert Schiele und Till Westmann: *Anatomy of a Native XML Base Management System.* VLDB Journal, 11(4):292–314, 2002.

[FKM01] Thorsten Fiebig, Carl-Christian Kanne und Guido Moerkotte: *Natix - ein natives XML-DBMS.* Datenbank-Spektrum, 1:5–13, 2001.

[Fou04a] The Apache Software Foundation: *Apache Xerces.* Online im Internet [Stand: September 2004], 2004. http://xml.apache.org/xerces2-j.

[Fou04b] The Apache Software Foundation: *The Apache XML Project: Xalan.* Online im Internet [Stand: September 2004], 2004. http://xml.apache.org/xalan-j.

[Fuh04] Norbert Fuhr: *INitiative for the Evaluation of XML Retrieval (INEX).* Online im Internet [Stand: September 2004], 2004. http://inex.is.informatik.uni-duisburg.de:2004.

[GHJV95] E. Gamma, R. Helm, R. Johnson und J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GJK+02] Sudipto Guha, H. V. Jagadish, Nick Koudas, Divesh Srivastava und Ting Yu: *Approximate XML Joins.* In: *Proc. of the ACM SIGMOD*, Seiten 287–298, 2002.

[GKT03] Torsten Grust, Maurice van Keulen und Jens Teubner: *Staircase Join: Teach a Relational DBMS to Watch its (Axis) Steps.* In: *Proc. of the VLDB*, Seiten 524–525, 2003.

[GNU] Free Software Foundation: *GNU Lesser General Public License. URL: http://www.gnu.org/copyleft/lesser.html.* Online im Internet [Stand: September 2004], 2002.

[Gra93] Goetz Graefe: *Query Evaluation Techniques for Large Databases.* ACM Computing Surveys, 25(2):73–170, 1993.

[Gra04] Jim Gray: *The Next Database Revolution.* In: Gerhard Weikum, Arnd Christian König und Stefan Dessloch (Herausgeber): *Proc. of the ACM SIGMOD*, Seiten 1–4. ACM, 2004.

[Gut84] Antonin Guttman: *R-Trees: A Dynamic Index Structure for Spatial Searching.* In: *Proc. of the ACM SIGMOD*, Seiten 47–57, 1984.

[GW97] Roy Goldman und Jennifer Widom: *DataGuides: Enabling Query Formulation and Optimization in Semistructured Databases.* In: *Proc. of the VLDB*, Seiten 436–445, 1997.

[HLM03] Philip J. Harding, Quanzhong Li und Bongki Moon: *XISS/R: XML Indexing and Storage System using RDBMS.* In: *Proc. of the VLDB*, Seiten 1073–1076, 2003.

[HM02] Elliotte R. Harold und W. Scott Means: *XML in a Nutshell.* O'Reilly & Associates, 2002.

[HMF99] G. Huck, I. Macherius und P. Fankhauser: *PDOM: Lightweight Persistency Support for the Document Object Model.* In: *OOPSLA Workshop „Java and Databases: Persistence Options", Denver, Colorado, USA*, November 1999.

[HR01] Theo Härder und Erhard Rahm: *Datenbanksysteme: Konzepte und Techniken der Implementierung, 2. Auflage.* Springer, 2001.

[HTM] *HTML Homepage des W3C.* Online im Internet [Stand: September 2004]. http://www.w3.org/MarkUp.

[IBM99] IBM: *OEM Hard Disk Drive Specifications for DNES-318350 / DNES-309170 (18/9 GB)*, 1999. Revision (1.2), S25L-1252-04.

[ISO] *ISO - International Organisation for Standardisation.* http://www.iso.org.

[JAKC+02] H. V. Jagadish, Shurug Al-Khalifa, Adriane Chapman, Laks V. S. Lakshmanan, Andrew Nierman, Stelios Paparizos, Jignesh M. Patel, Divesh Srivastava, Nuwee Wiwatwattana, Yuqing Wu und Cong Yu: *TIMBER: A native XML database.* VLDB Journal, 11(4):274–291, 2002.

[Jav04] *Java.* Online im Internet [Stand: September 2004], 2004. http://java.sun.com.

[JAX04a] *Java Architecture for XML Binding (JAXB).* Online im Internet [Stand: September 2004], 2004. http://java.sun.com/xml/jaxb.

[Jax04b] Jaxen Project Team: *Jaxen: Universal Java XPath Engine*, 2004. http://jaxen.codehaus.org.

[JDO04] *JDOM.* Online im Internet [Stand: September 2004], 2004. http://www.jdom.org.

[JTi01] *Project Info - JTidy*, 2001. http://sourceforge.net/projects/jtidy.

[Kan03] Carl-Christian Kanne: *Core Technologies for Native XML Database Management Systems.* Dissertation, Universität Mannheim, 2003.

[Kaw03] Kohsuke Kawaguchi: *Sun Multi-Schema XML Validator, Product Version: 1.2*, April 2003. http://wwws.sun.com/software/xml/developers/multischema.

[KM99] Carl-Christian Kanne und Guido Moerkotte: *Efficient Storage of XML Data.* Technischer Bericht, Universität Mannheim, 1999. http://pi3.informatik.uni-mannheim.de.

[KM00] Carl-Christian Kanne und Guido Moerkotte: *Efficient Storage of XML Data.* In: *Proc. of the ICDE*, Seite 198, 2000.

[KM03] Meike Klettke und Holger Meyer: *Speicherung von XML-Dokumenten - eine Klassifikation.* Datenbank-Spektrum, 5:40–50, 2003.

[KS04] Jürgen Krämer und Bernhard Seeger: *PIPES - A Public Infrastructure for Processing and Exploring Streams.* In: *Proc. of the ACM SIGMOD*, Seiten 925–926, 2004.

[KSBG02] Raghav Kaushik, Pradeep Shenoy, Philip Bohannon und Ehud Gudes: *Exploiting Local Similarity for Indexing Paths in Graph-Structured Data.* In: *Proc. of the ICDE*, Seiten 129–140, 2002.

[KYU01] Dao Dinh Kha, Masatoshi Yoshikawa und Shunsuke Uemura: *An XML Indexing Structure with Relative Region Coordinate.* In: *Proc. of the ICDE*, Seiten 313–320, 2001.

[LLD+02] Robert W. P. Luk, Hong Va Leong, Tharam S. Dillon, Alvin T. S. Chan, W. Bruce Croft und James Allan: *A Survey in Indexing and Searching XML Documents.* Journal of the American Society for Information Science and Technology, 53(6):415–437, 2002.

[LLG03] Chengfei Liu, Jixue Liu und Minyi Guo: *On Transformation to Redundancy Free XML Schema from Relational Database Schema.* In: *Web Technologies and Applications, APWeb*, Band 2642 der Reihe *Lecture Notes in Computer Science*, Seiten 35–46. Springer, 2003.

[Lom91] David B. Lomet: *Grow and Post Index Trees: Roles, Techniques and Future Potential.* SSD, Seiten 183–206, 1991.

[LS00] Hartmut Liefke und Dan Suciu: *XMILL: An Efficient Compressor for XML Data.* In: *Proc. of the ACM SIGMOD*, Seiten 153–164, 2000.

[LW04] Ingo Lütkebohle und Sebastian Wrede: *Catwalk. Einsatz des DBMS Berkeley DB XML.* iX. Magazin für professionelle Informationstechnik., 9:68–73, 2004.

[May03] Prof. Dr. Wolfgang May: *Vorlesung: Semistructured Data and XML*, 2003. may@informatik.uni-goettingen.de.

[McG01] David McGoveran: *The Age of the XML Database.* eAI Journal, 10:18–21, Oktober 2001. http://www.bijonline.com.

[MCS96] Mark L. McAuliffe, Michael J. Carey und Marvin H. Solomon: *Towards Effective and Efficient Free Space Management.* In: *Proc. of the ACM SIGMOD*, Seiten 389–400, 1996.

[Mic92] Microsoft Corporation: *Long Filename Specification, Version 0.5*, 1992. Hardware White Paper.

[Mic99] Microsoft Corporation: *FAT: General Overview of On-Disk Format, Version 1.02*, 1999. Hardware White Paper.

[Mic03] Sebastian Michel: *Verteilte Anfrageverarbeitung mit Webservices. Diplomarbeit. Philipps-Universität Marburg, Fachbereich Mathematik und Informatik*, April 2003.

[MLLA03] Xiaofeng Meng, Daofeng Luo, Mong-Li Lee und Jing An: *OrientStore: A Schema Based Native XML Storage System.* In: *Proc. of the VLDB*, Seiten 1057–1060, 2003.

[MWL+04] Xiaofeng Meng, Yu Wang, Daofeng Luo, Shichao Lu, Jing An, Yan Chen, Jianbo Ou und Yu Jiang: *OrientX : A Schemabased Native XML Database System.* Technischer Bericht, Information School Renmin University of China, Beijing, 100872, China, Juli 2004.

[Ora04] Oracle Technology Network: *Data Sheet: Oracle9i Index-Organized Tables.* Online im Internet [Stand: September 2004], 2004. http://www.oracle.com/technology/products/oracle9i/datasheets/iots/iot_ds.html.

[PAKC+03] Stelios Paparizos, Shurug Al-Khalifa, Adriane Chapman, H. V. Jagadish, Laks V. S. Lakshmanan, Andrew Nierman, Jignesh M. Patel, Divesh Srivastava, Nuwee Wiwatwattana, Yuqing Wu und Cong Yu: *TIMBER: A Native System for Querying XML.* In: *Proc. of the ACM SIGMOD*, Seite 672, 2003.

[Par04] Terence Parr: *ANTLR Parser Generator and Translator Generator.* Online im Internet [Stand: September 2004], 2004. http://www.antlr.org.

[Rel03] Relax NG Technical Committee: *RELAX NG home page.* Online im Internet [Stand: September 2004], 2003.

[Röh88] Lutz Röhrich: *Der Baum in der Volksliteratur, in Märchen, Mythen und Riten.* Adrien Finck and Gertrud Gréciano, Straßburg, 1988. http://www.maerchenlexikon.de/archiv/archiv/roehrich01.htm.

[Rus97] Mark Russinovich: *Inside the Windows NT Registry*, April 1997.

[Rus99] Mark Russinovich: *Inside the Registry*, Mai 1999. http://www.winntmag.com.

[RV03] Erhard Rahm und Gottfried Vossen: *Web & Datenbanken. Konzepte, Architekturen, Anwendungen.* dpunkt, 2003.

[SAX] *Simple API for XML (SAX).* Online im Internet [Stand: September 2004]. http://www.saxproject.org.

[SC01] Dieter Scheffner und Rainer Conrad: *Access Support Tree & Text Array: A Model for Physical Storage of XML Documents.* In: *GI Jahrestagung (1)*, Seiten 406–416, 2001.

[Sch99] Martin Schneider: *Parallele Verfahren zur Berechnung des räumlichen Verbunds. Diplomarbeit. Philipps-Universität Marburg, Fachbereich Mathematik und Informatik*, April 1999.

[Sch01] HARALD SCHÖNING: *Tamino - A DBMS designed for XML.* In: *Proc. of the ICDE*, Seiten 149–154, 2001.

[Sch03] HARALD SCHÖNING: *Tamino - A Database System Combining Text Retrieval and XML*, Seiten 77–89. Springer-Verlag Heidelberg, 2003.

[See03] MICHAEL SEEMANN: *Native XML-Datenbanken im Praxiseinsatz.* Software & Support Verlag GmbH, Frankfurt, 2003.

[SF02] DIETER SCHEFFNER und JOHANN-CHRISTOPH FREYTAG: *The XML Query Execution Engine (XEE).* In: HELE-MAI HAAV und AHTO KALJA (Herausgeber): *Proceedings of the Baltic Conference, BalticDB&IS 2002, Volume 1.* Institute of Cybernetics at Tallin Technical University, 2002.

[SF04] JÉRÔME SIMÉON und MARY FERNÁNDEZ: *Galax - an Implementation of XQuery.* Online im Internet [Stand: September 2004], 2004. http://db.bell-labs.com/galax.

[SH03a] ARNE SCHÄPERS und RUDOLF HUTTARY: *Daniel Düsentrieb. C#, Java, C++ und Delphi im Effizienztest, Teil 1.* c't, 19:204 ff, 2003.

[SH03b] ARNE SCHÄPERS und RUDOLF HUTTARY: *Daniel Düsentrieb. C#, Java, C++ und Delphi im Effizienztest, Teil 2.* c't, 21:222 ff, 2003.

[Sho95] SHORE TEAM: *SHORE: Combining the Best Features of OODBMS and File Systems.* In: MICHAEL J. CAREY und DONOVAN A. SCHNEIDER (Herausgeber): *Proc. of the ACM SIGMOD*, Seite 486. ACM Press, 1995.

[SKWW01] ALBRECHT SCHMIDT, MARTIN L. KERSTEN, MENZO WINDHOUWER und FLORIAN WAAS: *Efficient Relational Storage and Retrieval of XML Documents.* In: *WebDB (Selected Papers)*, Band

1997 der Reihe *Lecture Notes in Computer Science*, Seiten 137–150. Springer, 2001.

[SMFH01] Mehul A. Shah, Samuel Madden, Michael J. Franklin und Joseph M. Hellerstein: *Java Support for Data-Intensive Systems: Experiences Building the Telegraph Dataflow System.* SIGMOD Rec., 30(4):103–114, 2001.

[SOA04] *SOAP Version 1.2, W3C Recommendation 24 June 2003.* Online im Internet [Stand: September 2004], 2004. http://www.w3.org/TR/soap12-part1.

[SS04] Andreas Stiller und Herbert Schmid: *Intels V8. Intels neue C/C++-Compiler für Windows und Linux.* c't, 3:160 ff, 2004.

[SSD04] *Solid State Disk.* Online im Internet [Stand: September 2004], 2004. http://www.webopedia.com/TERM/S/solid_state_disk.html.

[Sta01] Kimbro Staken: *Introduction to Native XML Databases.* Online im Internet [Stand: September 2004], 2001. http://www.xml.com/lpt/a/2001/10/31/nativexmldb.html.

[Sta02] Kimbro Staken: *An Introduction to the XML:DB API.* Online im Internet [Stand: September 2004], 2002. http://www.xml.com/pub/a/2002/01/09/xmldb_api.html.

[STW01] Sergej Sizov, Anja Theobald und Gerhard Weikum: *Ähnlichkeitssuche auf XML-Daten.* In: *GI-Fachtagung für Datenbanksysteme in Business, Technologie und Web*, Seiten 364–383, 2001.

[STZ+99] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt und Jeffrey F.

NAUGHTON: *Relational Databases for Querying XML Documents: Limitations and Opportunities.* In: *Proc. of the VLDB*, Seiten 302–314, 1999.

[Suc01] DAN SUCIU: *On Database Theory and XML.* SIGMOD Rec., 30(3):39–45, 2001.

[Sun02] SUN MICROSYSTEMS: *The Java HotSpot Virtual Machine, v.1.4.1, d2*, 2002. http://java.sun.com.

[SW00] HARALD SCHÖNING und JÜRGEN WÄSCH: *Tamino - An Internet Database System.* Seiten 383–387, 2000.

[SWK+02] A. R. SCHMIDT, F. WAAS, M. L. KERSTEN, M. J. CAREY, I. MANOLESCU und R. BUSSE: *XMark: A Benchmark for XML Data Management.* In: *Proc. of the VLDB*, Seiten 974–985, Hong Kong, China, August 2002.

[TDCZ02] FENG TIAN, DAVID J. DEWITT, JIANJUN CHEN und CHUN ZHANG: *The Design and Performance Evaluation of Alternative XML Storage Strategies.* SIGMOD Rec., 31(1):5–10, 2002.

[TFW03] THOMAS TESCH, PETER FANKHAUSER und TIM WEITZEL: *Skalierbare Verarbeitung von XML mit Infonyte-DB.* In: *GI-Fachtagung für Datenbanksysteme in Business, Technologie und Web*, Seiten 578–590, 2003.

[Wal98] NORMAN WALSH: *A Technical Introduction to XML*, 1998. http://www.xml.com/pub/a/98/10/guide0.html.

[Wei84] REINHOLD P. WEICKER: *Dhrystone: A Synthetic Systems Programming Benchmark.* In: *Communications of the ACM*, Band 27, Seiten 1013–1030, October 1984.

[WNL03] KHIN-MYO WIN, WEE KEONG NG und EE-PENG LIM: *ENA-XS: Efficient Native XML Storage System.* In: *Web Technologies and Applications: 5th Asia-Pacific Web Conference, APWeb*

2003, Xian, China, April 23-25, 2003., Seiten 59–70. Springer-Verlag Heidelberg, 2003.

[WS004] *Web Services Description Language (WSDL) Version 2.0, W3C Working Draft.* Online im Internet [Stand: September 2004], August 2004. http://www.w3.org/TR/wsdl20.

[Xin04] *Apache Xindice.* Online im Internet [Stand: September 2004], 2004. http://xml.apache.org/xindice.

[XML98a] *W3C Recommendation XML 1.0*, 1998. http://www.w3.org/TR/REC-xml.

[XML98b] *XML in Ten Points*, 1998. http://www.w3.org/XML/1999/XML-in-10-points.html.en.

[XML04] *W3C Recommendation XML 1.1*, 2004. http://www.w3.org/TR/2004/REC-xml11-20040204.

[XPa99] *XML Path language (XPath), Version 1.0, W3C Recommendation.* Online im Internet [Stand: September 2004], 1999. http://www.w3.org/TR/xpath.

[XPa03] *XML Path language (XPath), Version 2.0, W3C Working Draft.* Online im Internet [Stand: September 2004], 2003. http://www.w3.org/TR/xpath20.

[XQu03a] *XQuery 1.0: An XML Query Language.* Online im Internet [Stand: September 2004], 2003. World Wide Web Consortium. http://www.w3.org/TR/xquery.

[XQu03b] *XQuery Use Cases. W3C Working Draft, 12 Nov. 2003*, 2003. World Wide Web Consortium. http://www.w3.org/TR/xquery-use-cases.

[XQu04] *XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Working Draft, 20 Feb. 2004*, 2004. World Wide Web Consortium. http://www.w3.org/TR/xquery-semantics.

[XSD] *XML Schema Standard des W3C.* Online im Internet [Stand: September 2004]. http://www.w3.org/XML/Schema.

[XSL01] *Extensible Stylesheet Language (XSL) Version 1.0.* Online im Internet [Stand: September 2004], 2001. http://www.w3.org/TR/xsl.

[XXL] Philipps University of Marburg: The Database Research Group: *XXL - eXtensible and fleXible Library. URL: http://www.xxl-library.de*, 2004.

[Xyl01] Lucie Xyleme: *Xyleme: A Dynamic Warehouse for XML Data of the Web.* Seiten 3–7, 2001.

[YAA03] Hailing Yu, Divyakant Agrawal und Amr El Abbadi: *Tabular Placement of Relational Data on MEMS-based Storage Devices.* Seiten 680–693, 2003.

[ZS97] Kaizhong Zhang und Dennis Shasha: *Tree pattern matching.* Seiten 341–371, 1997.

Abbildungsverzeichnis

Tabellenverzeichnis

Danksagungen

Ich möchte mich besonders bei meinem Anleiter, Herrn Prof. Dr. Bernhard Seeger, für die freundliche und engagierte Betreuung der Arbeit bedanken. Obwohl das von mir gewählte Thema nicht in seinem bisherigen Forschungsschwerpunkt lag, stand er mir immer mit wertvollen Ratschlägen zur Seite.

Am Gesamtprojekt haben viele Personen mitgewirkt, bei denen ich mich ganz herzlich bedanken möchte. Besonders erwähnen möchte ich meine Kollegen aus der Arbeitsgruppe Michael Cammert, Christoph Heinz, Jürgen Krämer und Tobias Schäfer, die alle mit viel Einsatz das Projekt XXL voranbringen. Teile des Quelltexts entstanden ursprünglich im Rahmen von Fortgeschrittenenpraktika. Mein besonderer Dank gilt hier Sebastian Michel, Martin Kaletsch, Lars Kuckenburg, Juri Bender und Andrej Rodionov.

Für das Korrekturlesen bedanke ich mich ganz herzlich bei meiner Partnerin Annette Baisch, bei meinen Eltern, bei Yvonne Stöter und bei Sebastian Michel.

Für die seelische Unterstützung und für viel Liebe bedanke ich mich sehr bei meiner lieben Annette. Sie hat mir zusätzlich durch die Freistellung von alltäglichen Arbeiten sehr dabei geholfen, das gewünschte Abgabedatum einhalten zu können.

Ohne alle diese Hilfen wäre vieles nicht realisierbar gewesen. Vielen Dank.

Marburg im Oktober 2004

Index